FH Vorarlberg
University of Applied Sciences

Waste object classification with AI on the edge accelerators

Master Thesis

Submitted in Fulfillment of the Degree

Master of Science in Engineering, MSc

Vorarlberg University of Applied Sciences

Master's in Mechatronics

Support Vorarlberg University of Applied Sciences

Prof. (FH) Dipl.-Ing. Amann Robert

Support Kasetsart University at Faculty of Engineering

Assoc. Prof. Dr. Chowarit Mitsantisuk

Handed in by

Schneider Michael, BSc

51835259

Dornbirn, 31.12.2020

Kurzreferat

Klassifizieren von Abfallobjekten mit Edge KI Prozessoren

Das Klassifizieren von Abfall mit Neuronalen Netzen ist bereits Thema in einigen wissenschaftlichen Arbeiten. Eine Anwendung im Embedded Systems Bereich mit aktuellen KI-Prozessoren zur Beschleunigung der Inferenze wurde dabei noch nicht behandelt. In dieser Masterarbeit wird ein Prototyp erstellt der Müllobjekte klassifiziert und den für das Objekt zutreffende Behälter automatisch öffnet. Der Einsatzbereich liegt dabei auf dem öffentlichem Raum.

Für die Klassifizierung wird ein Datenset mit 25,681 Bilder und 11 Klassen erstellt um die Convolutional Neuronal Networks EfficientNet-B0, MobileNet-v2 und NASNet-Mobile neu zu trainieren. Diese Convolutional Neuronal Network werden auf den aktuellen Edge KI-Prozessoren von Google, Intel und Nvidia ausgeführt und auf Leistung, Verbrauch und Genauigkeit verglichen.

Die Masterarbeit evaluiert das Ergebnis dieses Vergleiches und zeigt dabei die Vor- und Nachteile der jeweiligen Prozessoren sowie der Convolutional Neuronal Networks. Für den Prototpen wird die am besten geeignete Kombination aus Hardware und KI-Architekture verwendet und auf der Universitätsmesse KasetFair2020 ausgestellt. Dabei wird eine Meinungsumfrage zur Anwendung der Maschine durchgeführt.

Abstract

Waste object classification with AI on the edge accelerators

The classification of waste with neural networks is already a topic in some scientific papers. An application in the embedded systems area with current AI processors to accelerate the inference has not yet been discussed. In this master work a prototype is created which classifies waste objects and automatically opens the appropriate container for the object. The area of application is in the public space.

For the classification a dataset with 25,681 images and 11 classes is created to re-train the Convolutional Neuronal Networks EfficientNet-B0, MobileNet-v2 and NASNet-Mobile. These Convolutional Neuronal Network run on the current Edge AI processors from Google, Intel and Nvidia and are compared for performance, consumption and accuracy.

The master thesis evaluates the result of these comparisons and shows the advantages and disadvantages of the respective processors and the Convolutional Neuronal Networks. For the prototype, the most suitable combination of hardware and AI architecture is used and exhibited at the university fair KasetFair2020. An opinion survey on the application of the machine is conducted.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI Artificial Intelligence. III, 3, 4, 8, 45, 97, 98

ASIC Application Specific Integrated Circuit. 9

CISC Complex Instruction Set Computer. 16

CNN Convolutional Neuronal Network. II–IV, XI, 3–9, 11, 14, 21–31, 33, 41, 43, 45, 49, 52, 54, 63, 67, 69, 70, 72, 74, 76, 78, 80, 81, 83–85, 87, 89–92, 95, 97, 98

CPU Central Processing Unit. 10, 13–16, 48, 49, 53, 54, 67, 69, 78

CUA Coral USB Accelerator. VIII, IX, XI, 3, 13, 14, 41, 52, 54–64, 66, 78–80, 89–91, 97, 98

EffNet EfficientNet-B0. II, III, VIII, IX, 3, 21, 28, 32, 33, 35, 38–42, 47, 48, 50–52, 54, 67–76, 78, 89, 97, 98

GFLOPS Giga Floating Point Operations Per Second. 13

GPU Graphics Processing Unit. IV, 3, 10, 15, 17, 32, 33, 35, 39, 41, 43, 49, 57, 62, 63, 65, 67, 69, 70, 72–74, 76, 78, 80, 81, 83, 89, 97

IR Intermediate Representation. 50, 98

MAC Multiply Accumulate Operation. 16

MAD Multiply-Add. 16

MNV2 MobileNet-v2. II, III, VIII, IX, 3, 21, 22, 26, 32, 33, 35, 36, 39, 41, 48, 50–52, 54–65, 67–89, 91, 92, 97, 98

MXU Matrix Multiply Unit. 16

Nano Jetson Nano. IX, XI, 3, 68, 69, 71–73, 75, 77, 89, 90, 97, 98

NAS Neural Architecture Search. 26

NAS-M NASNet-Mobile. II, III, VIII, IX, 3, 21, 32, 33, 35, 37, 39–42, 48–52, 54–65, 67–76, 78–87, 89, 97

NCS2 Neuronal Compute Stick 2. V, X, XI, 3, 13, 14, 17, 45, 52, 78–90, 97, 98

RPI4 Raspberry Pi 4. 13, 54, 61, 78, 80, 91

SHAVE Streaming Hybrid Architecture Vectore Engine. VII, 17, 18

SVM Support Vector Maschine. 6

TF Tensorflow. 50

TOPS Tera Operations Per Second. 13, 16

TPU Tensor Processing Unit. IV, 3, 13, 14, 16, 17, 32, 33, 39, 41, 43, 47, 48, 54, 57, 61–63, 65, 67, 70, 72–74, 76, 78, 80, 81, 83, 89, 97

VPU Vision Processing Unit. IV, 3, 13, 14, 17, 97

Summary

In this thesis a concept is developed that classifies waste objects via a video stream. The classification is done with a Convolutional Neuronal Network on a AI accelerator. The system should have a high accuracy and low power consumption. To achieve this, an AI system specially developed for the application in embedded systems is used. For this very current field of application new technologies and computing units have been developed, of which 3 are described and compared in this thesis. The Edge-AI devices used are the Coral USB Accelerator developed by Google with the edge Tensor Processing Unit, Nvidia's smallest developer board Jetson Nano with an integrated Tegra Embedded Graphics Processing Unit and Intels Neuronal Compute Stick 2 which uses a Myriad X Vision Processing Unit developed by Movidius. For the classification of images the Convolutional Neuronal Networks are the state of the art network architectures. In recent years, newer networks have been developed, especially in the mobile area, which require significantly less resources than the standard CNN models. The neural network model is the main factor for accuracy and plays an important role in performance. For the concept 3 CNN models developed for mobile use are trained and tested with the data set of the waste objects. For the training the Tensorflow/Keras framework is used. The models are thus the Keras pre-trained mobile CNNs MobileNet-v2, NASNet-Mobile and EfficientNet-B0. In co-operation with the Faculty of Environmental Technology, 10 categories of waste have been identified, covering the majority of public waste. From these 10 categories, a data set of 25,681 samples was created which is divided into 70% training, 20% validation and 10% testing. The models were trained with the training and validation set on Google Colab platform with the cloud GPU and TPU respectively. This is to detect if the AI accelerator used in the training has an effect on the result of the CNN inference. The test data was used to create confusion matrices from the Keras models to determine the quality. The Keras models were then compiled and optimized for the respective platforms of the end devices. The compiled models are used to run an experiment on the Edge devices. The 2667 images of the test data set are classified by the models. During this process CPU and memory usage, inference time and power consumption were measured. Based on the findings of the results, a prototype was developed to test the concept under real conditions. This prototype was exhibited at the university fair Kasetfair2020. In co-operation with the Faculty of Environmental Engineering an opinion survey about AI in Reycling was conducted.

1 Introduction

This thesis is part of the Master Program "Mechatronics" at the FH Vorarlberg University of Applied Sciences and is written during a study abroad stay at the Kasetsart University in Bangkok, Thailand.

One of the biggest problems this time is the global climate change. Part of this problem is global resource consumption and waste management. The Open World Bank classifies Thailand as a country with an upper middle income economy. According to the book "What a Waste 2.0: A Global Snapshot of Solid Waste Management to 2050" from the Open World Bank upper-middle income countries, like Thailand, have a recycling rate of 4% and 30% of the waste gets open dumb. As an East Asian country, Thailand is part of the region that generate the most waste. Kaza et al. 2018

According to a 2017 study by Chualongkorn University in Bangkok, 34% of Bangkok residents do not separate their waste. The reasons for this are: no nearby waste collection point (4%); no vehicle to transport the waste (4%); low sales value of recyclable waste (5%); no knowledge about how to recycle (9%); no matter because it is remixed (14%); no interest, no time (20%); no space for the waste (20%) and too few waste containers (22%). Vassanadumrongdee and Kittipongvises 2018

On the basis of the Pollution Control Department Report 2018 of Thailand, 27% of Thailands municipal solid waste of 2108 or 7.36 million tons where not disposed appropriatly. Also 2,881 local administrative organizations have no proper solid waste management. *Booklet on Thailand State of Pollution* 2019

Due to the rapid increase of the computing power of processors is it now possible to use Artificial Intelligence in various areas. This thesis deals specifically with the area of AI in embedded systems or rather confessed AI on the Edge.
Artificial Intelligence on the Edge is a very hot topic as it works without an Internet connection. This results in a higher data security and since no data transfer is necessary, higher speed is achieved. Further advantages are the low power consumption and the better scalability. Due to the better real-time processing it is a key technology in the area of Internet of Things and Industry 4.0. It is therefore not surprising that all major processor manufacturers such as Google, Intel or Nvidia have already launched hardware in this area.

This thesis will connect the problem with waste management and the AI on the Edge technology. There the Edge AI is used to solve a classification problem with images. For this we use different Convolutional Neuronal Network in the mobile

area, Efficientnet-B0, Mobilenet-v2 and NASNetMobile. These are pre-trained CNN models for edge devices that are available in the Tensorflow-Keras framework that is used in this thesis. The data set to re-train the models contain waste images from the public sector and is created in co-operation with the Faculty of Environmental Engineering of Kasetsart University.

As hardware the three devices from Google, Intel and Nvidia are used. These use different processors for parallel processing with different advantages and disadvantages. In an experiment the three CNN models are executed on these edge devices and tested for accuracy, speed, power consumption, CPU and memory usage.

Based on the gained knowledge, a prototype is created which takes pictures in real time with the help of a camera and classifies them. The result of the classification is used to automatically open the correct waste bin.

2 Literature Review

A considerable amount of research has been and is still being done in the field of artificial intelligence, especially in the area of image processing. The classification of objects is a method with a wide range of applications. The scientific work mentioned here refers to the recognition and classification of waste objects.

In the work Bircanoğlu et al. 2018 an Convolutional Neuronal Network architecture is developed called RecycleNet. The developed model is compared with the most common CNNs MobileNet, DenseNet and ResNet for accuracy and speed. The inference of the models is executed on a PC, whereby the performance of the execution is benchmarked between the Xeon CPU with 2.20 GHz and the GTX980 GPU.

The research project of George E. Sakr et al. Sakr et al. 2016 compares the classification of waste objects by a CNN and a Support Vector Maschine for an autonomous recycling machine. For the comparison the AlexNet Convolutional Neuronal Network is used. The technology with the higher accuracy is listed on a Raspberry Pi3 without AI accelerator which uses LEDs to assign the correct container.

In the article Costa et al. 2018 an experiment is conducted which compares CNNs like VGG16 and AlexNet, as well as classifiers like SVM, K-Nearest Neighbor and Random Forest for accuracy. The objects are classified in 4 classes: glass, metal, paper and plastic.

The topic of waste object classification with convolution neuronal networks is also treated by Yinghao Chu et al. in the research article Chu et al. 2018. The classification is based on a multilayer hybrid system consisting of a CNN, the in 2012 published AlexNet, and a multilayer perceptron. The waste objects is positioned and rotated in a dark grey box, using a high resolution camera. Furthermore, sensors are used to measure the weight and detect whether the object is metallic. The object information is collected by a PC and evaluated by the multilayer hybrid system and classifies the object to recyclable or others.

Another field of application is the detection of waste on waste trays as used in fast food chains. This topic is dealt with in the research work of Sousa, Rebelo, and Cardoso 2019. It deals with the hierarchical deep learning application of image recognition and classification. The work uses the Faster R-CNN for this purpose.

A comparison between the CNNs ResNet50, VGG16 and ResNet18 is made in the experiment from the work of Dipesh Gyawali et al. Gyawali et al. 2020. In this work the CNNs are trained on the 4 classes plastic, paper, metal and glass and compared for their accuracy.

The current research in the field of waste object classification is concerned with excluding the comparison of different Convolutional Neuronal Network architectures or other machine classification methods. The accuracy is the only factor that is compared in these studies. A comparison of CNNs designed for embedded systems is still open for waste classification. Furthermore, the use of AI accelerators has not yet been considered in the research work, which is a focus of this thesis.

3 Concepts

This chapter discuss the concept of this master thesis. The idea is to create a prototype that acts as a recycling assistant to help people to dispose their waste in the right containers. The field of application is limited to the public space, otherwise it would go beyond the scope of this master thesis. Prerequisites for the prototype are independence of location, highest possible accuracy and classification speed.

The garbage is captured with a video stream and each frame serves as input for Convolutional Neuronal Network, which then predicts what type of waste it is. The CNN is executed by Artificial Intelligence on the Edge. Depending on the prediction of the CNN, the waste container intended for this class, is opened. The concept is illustrate in the diagram 3.1

AI on the Edge is on of the most current topic. It offers, in contrast to the cloud based solution, the possibility to run AI systems without internet connection. The data is evaluated directly at the end device, which results in a significantly lower latency time. Since the data is not transmitted via the web, these systems offer a higher data security. Edge-AI's are designed for use with embedded systems, therefore they consume less power than a server based solution. Lee, Tsung, and Wu 2018

Chip manufacturers like Intel (section 3.3) and Nvidia (section 3.2) already offer versions of their processors for edge computing. Also Google (section 3.1) offers its 2016 processor designed specifically for machine learning of neural networks as an edge version. For the concept, the development versions of these manufacturers are compared with each other. The respective advantages and disadvantages of each processor, as well as their handling, are discussed.

Since edge hardware is generally embedded systems, its performance and memory is often limited. Therefore, CNN models have been developed for this application area. The various models differ in accuracy, speed and memory consumption.
To manipulate and train the models of neural networks the Tensorflow/Keras Framework is used. This framework is developed by Google Brain, the Deep Learning department of Google LCC. This framework is used because, according to the manufacturers, it is possible to compile Tensorflow model to the respective end devices. Furthermore the Keras framework for the mobile area offers 3 different pre-trained CNNs EfficientNet, MobileNet-v2 and NASNet-Mobile. The 3 models are executed on the Edge processors and compared on CPU and memory usage, accuracy, speed and power consumption.

From the result of the comparison, the most suitable combination of CNN and Edge devices is used for the prototype.

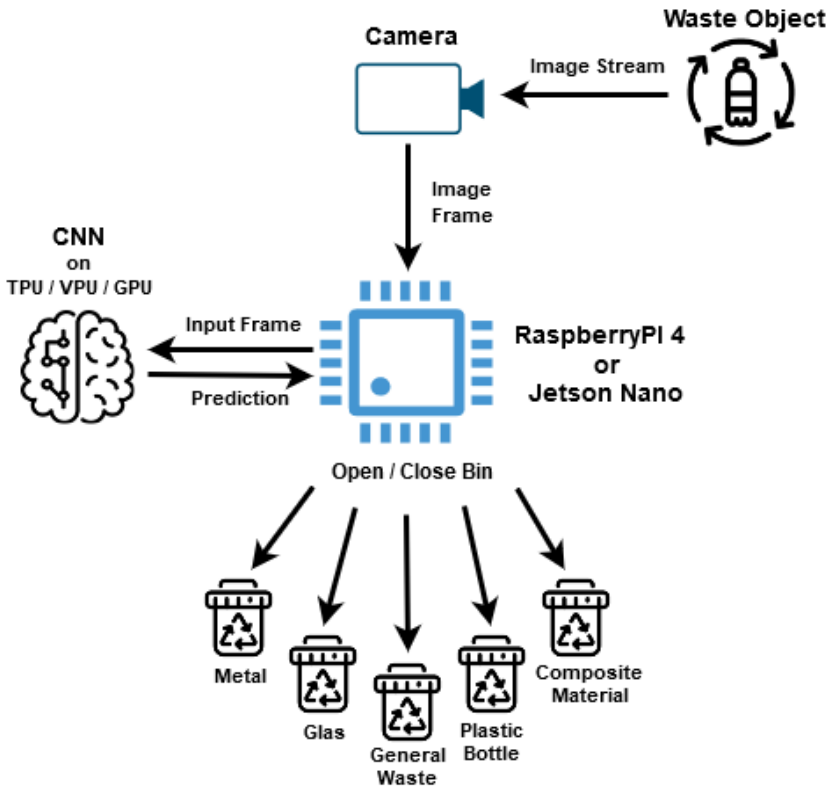


Figure 3.1: Overview diagram of the concept

3.1 Google Coral USB Accelerator

Coral is the product range of Google's edge TPUs. The external USB version is used in this case, which interacts with a Raspberry Pi 4 via the USB interface. At the time of writing the edge TPU was available as external version aIs SoM (System on Modul), PCIe accelerator, M.2 accelerator A+E and B+M or as a developer board. There could be performance differences between these variants, which will not be discussed further. More information about the developer board can be read in the spectrum IEEE review paper Cass 2019

Google released its Tensor Processing Unit at its own in-house exhibition in 2016, and has been using the technology at their data center for a year. The unit is specifically designed for machine learning applications of neural networks (ASIC). In year 2018, Google made the 2 version of the TPU for cloud computing available and the embedded version has been commercially available since 2019.

The heart of a TPU is the 8-bit matrix multiplication unit which consists of NxN multiplication accumulators that perform 8-bit multiplications. The product of the multiplication is summed up and stored in an accumulator. To save energy when reading the SRAM, a systolic array is used, reducing the number of read and write commands. Furthermore, the TPU acts as a co-processor via the PCIe interface. Inference Models are loaded into the TPU's memory completely if possible to avoid interaction with the host CPU and save time during deployment. The TPU uses a CISC instruction set to facilitate hardware design and debugging. N. Jouppi et al. 2018; N. P. Jouppi et al. 2017

3.2 Nvidia Jetson Nano

Nvidia offers their own platform for edge computing with their Jetson. The Jetson Nano is the smallest Graphics Processing Unit device in this series with 128 cores. Nvidia has developed its own Linux operating system for the Jetson series, the "Linux4Tegra". It is a Ubuntu 18.4 version specially adapted for machine learning and parallel computing. In this thesis the developer board will be used but also a module version with the same specification can be purchased.

GPUs have been used in the visual area of computers since the mid 1990s. In contrast to conventional CPUs, which processes tasks very quickly and sequentially, GPU are designed for parallel processing. Modern GPUs are machines for massive parallel computations and are therefore suitable for training AI. A GPU consists of many cores that can be used individually. In order to access them, the programming technique CUDA developed by NVIDIA is used. Baji 2018; Nickolls and Dally 2010 ; Garland et al. 2008

3.3 Intel Neuronal Compute Stick 2

In 2016 Intel takes over the chip manufacturer Movidius, which has specialized in low-power processors in the field of machine vision. From this takeover Intel developed the Myriad X VPU, which is also built into the NCS2. Vision Processing Units are co-processors that are specially designed to perform machine vision tasks in the AI area. Myriad X contains 16 programmable SHAVE cores and a dedicated neural engine to accelerate the inference of neural networks.

Like the GPU, the VPU has multiple cores and is designed for high parallelization tasks. However, the VPU differs from the GPU in its special purpose. GPUs usually still contain hardware for the application of 3D graphics which is missing in the VPU. Also the memory architecture of the GPU is designed for bitmap images. Barry, Brick, et al. 2015; Barry, Connor, et al. 2015; Libutti et al. 2020 Rivas-Gomez et al. 2018;

4 Waste Categories and Materials

In order to solve the classification problem, the pre-trained CNN will be re-trained. For this purpose, a data set is created that contains the images of the objects to be classified. For this master thesis, the data set is divided into 10 categories, see figures at 4.2, that cover most of the public waste. Explicit attention was also paid to the fact that the categories differ in form, color and transparency to facilitate classification by the CNN. This subdivision was made in agreement with Assist.Prof.Cheema Soralump from the Faculty of Environment Engineering. This selection also covers most of the waste thrown into the sea. These are, according to the Pollution Control Department Report 2019 from Thailand, plastic bags (18.9%), plastic bottles (8.6%), thin plastic shopping bags (8.4%), foam dishes and bowls (6.9%), glass bottles (6.6%), food and snack packages (6.1%), straw and swizzle sticks (4.6%), foam scraps (4.4%), foam meal boxes (3.8%) and plastic cups (3.6%). The remaining 28.1% consists of various types of waste. *Booklet on Thailand State of Pollution* 2019
A general overview of the waste flow in Thailand is seen in figure 4.1

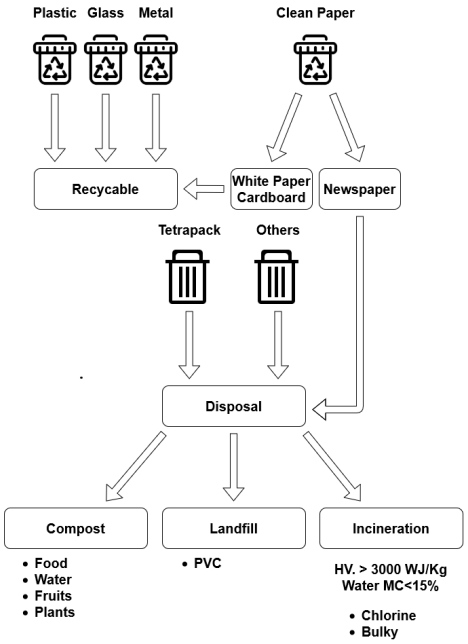


Figure 4.1: Overview of the waste flow in Bangkok
Source: Assist.Prof.Cheema Soralump

(a) Example image of the food bowl class

(b) Example image of the food box class

(c) Example image of the glas bottle class

(d) Example image of metal can class

(e) Example image of plastic bag class

(f) Example image of plastic bottle class

(g) Example image of plastic cup class

(h) Example image of plastic cutlery class

(i) Example image of snack wrap class

(j) Example image of tetrapack class

Figure 4.2: Example images of the 10 used categories

5 Hardware Overview

In this chapter the technical specification of the hardware is explained in more detail. An overview hardware specifications of the main- or host-CPU with the AI accelerator system is provided in the table 5.1. The advantages and disadvantages of AI acceleration processors and their design are considered. A general comparison of this devices was done by Nokia Bell Lab at a AI and IoT challenge 2019 in New York and the result can be read in the paper Antonini et al. 2019

At the end of this chapter are the benchmark test from Coral 5.3 and Nvidia 5.4 illustrated. It should give an overview of the inference speed of the used hardware devices and the supported neuronal network architectures.

Hardwarespecifications			
Type	RPI4 + Intel NCS2	RPI4 + GoogleCUA	Nvidia Jetson Nano
CPU	Quad-core ARM Cortex-A72 64-bit @ 1.5 Ghz	Quad-core ARM Cortex-A72 64-bit @ 1.5 Ghz	Quad-Core ARM Cortex-A57 64-bit @ 1.42 Ghz
RAM	2GB LPDDR4	4GB LPDDR4	4GB LPDDR4
AI accelerator	(extern) Intel Myriad X VPU	(extern) Google Edge TPU coprocessor	(intern) 128-core Maxwell
Host-Interface	USB 3.0	USB 3.0	PCIe
Spezified Performance	4 TOPS	4 TOPS	472 GFLOPS
Spezified Consumption	ca. 4 W (1W VPU + 3W RPI4)	ca. 5 W (1W/2W TPU + 3 RPI4)	ca.5W
Advantages	lower power consumption large intern memory	lower power consumption fast inference many modules	higher flexability higher accuracy
Disadvantages	slow inference lower flexability	lower flexability small intern memory	shared memory high power consumption

Table 5.1: Hardware specification tablehardware specifications of the used configurations of AI accelerators
Source: https://developer.nvidia.com/embedded/jetson-nano-developer-kit
https://www.intel.ai/intel-movidius-myriad-vpus
https://coral.ai/products/accelerator

5.1 USB Accelerator

The Edge TPU and the Myriad VPU are used here as USB accelerators. These are external co-processors that communicate with the host CPU via the USB interface. USB 3.0 is recommended for both devices, the lower data transfer rate with USB 2.0 can become a bottleneck and thus reduce performance. The table 5.2 shows the performance difference of the inference between the two processors on the MobileNet-v2 SDD, which is a object recognition CNN. A huge gap in performance can be seen in the Coral edge TPU, with USB 3.0, where the inference is about 5 times faster than with USB 2.0. With a 30% faster inference the Myriad VPU performs significantly better with USB 3.0 than with USB 2.0. Similar values are obtained in the experiment from the paper Antonini et al. 2019.

As co-processors, these USB accelerators have their own computing unit as well as their own memory and therefore put less strain on the resources of the main CPU. This allows several devices to be used in parallel on one host. A disadvantage is the lower transfer speed of the USB interface compared to the PCIe bus.

Setup	**Raspberry Pi 4 MobileNet-v2 SDD (ms)**
Google CUA USB2.0	102,3
Google CUA USB3.0	18,2
Intel NCS2 USB2.0	116,7
Intel NCS2 USB3.0	80,4

Table 5.2: Benchmark between USB2.0 and USB3.0 of Intel NCS2 and Google CUA Source: https://www.hackster.io/news/benchmarking-the-intel-neural-compute-stick-on-the-new-raspberry-pi-4-model-b-e419393f2f97

5.2 Jetson Nano - Graphics Processing Unit

The GPU is another programmable processor besides the CPU. While a CPU consists of 1 or few cores, a GPU has thousands of cores. Originally, the GPU was developed for displaying 3D graphics on a computer. Nowadays the cores can be programmed directly, for example with the CUDA framework from Nvidia, which makes it possible to perform massive parallel tasks on a GPU. Therefore, this technology is very much used in the field of AI. Because of the enormous number of operations that are executed in parallel during training and with a CPU, this cannot be done in a reasonable amount of time. Garland et al. 2008; Baji 2018

The Jetson Nano uses a 128 core Maxwell GPU as AI accelerator. Compared to the TPU and VPU, the Nano is much more flexible, because interger and floating point can be calculated. It also supports all common platforms. Most deep neural network models can be executed directly, but with a lower performance. For the ideal use Nvidia's TensorRT engine should be used. The Nano can also be used to train smaller neural networks. The major drawbacks of the embedded GPU are higher power consumption and the shared 4GB RAM memory with the CPU. This has the consequence that the GPU quickly runs out of memory when other programs are running.

5.3 Coral - Tensor Processing Unit

The TPU is a processor specifically designed to train and inference neural networks. The basic functions required for this are multiplication and addition of numbers. To perform these functions as quickly as possible, the TPU has a Matrix Multiply Unit. The first version of the TPU consisted of 256x256 8-bit integer multiply accumulates (MACs or MADs) with a clock frequency of 700MHz. In addition, the processor contains arithmetic units for activation and pooling, a unified buffer to store intermediate values of operations and a bus system for fast data transfer. For communication with the host CPU a CISC instruction set us which is aligned to Googles AI framework Tensorflow. The block diagram of v1 TPU is shown in figure 5.1. N. Jouppi et al. 2018; N. P. Jouppi et al. 2017; Ross et al. 2016

The Edge-TPU is a significantly downsized version of the original TPU, whose construction has not yet been published. From the performance data of 4 TOPS it can be assumed that it has a 64x64 8-bit integer MXU. The application area of the Coral is limited to the execution of 8-bit quantized neural networks. A disadvantage is that only models from the Tensorflow framework are supported by the CISC instruction set. Libutti et al. 2020; Jacob et al. 2018

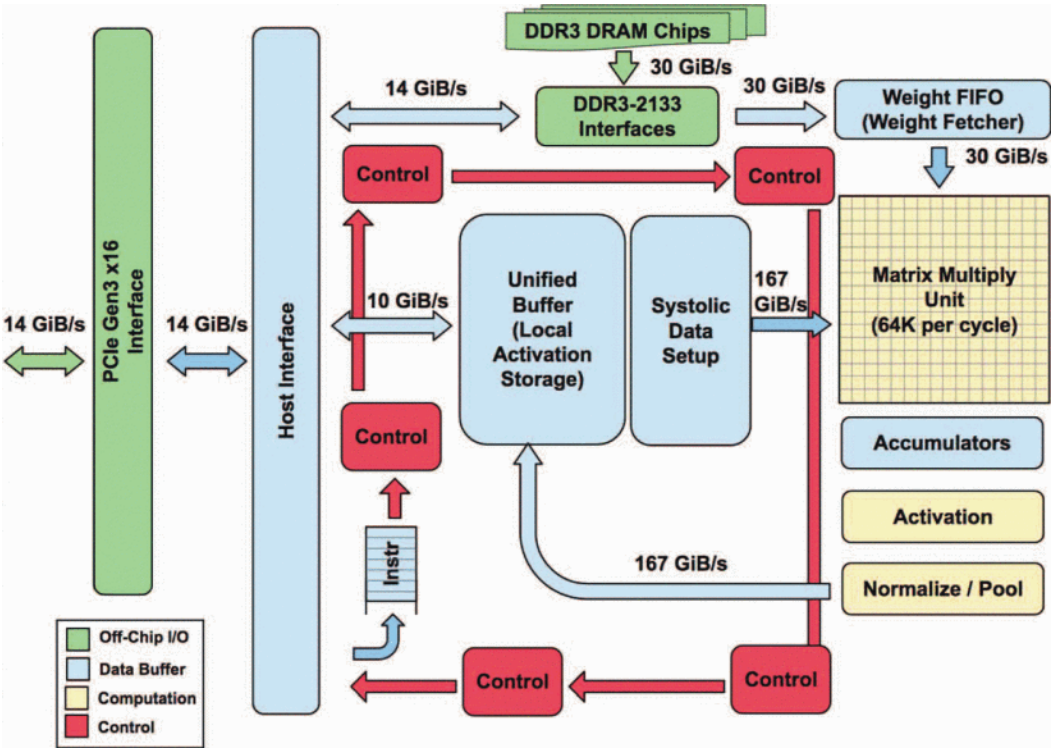


Figure 5.1: Block diagram of the first version of Googles TPU
Source: N. Jouppi et al. 2018

5.4 Intel Neuronal Compute Stick 2 - Vision Processing Unit

A Myriad X Vision Processing Unit is used in the Intel Neuronal Compute Stick 2, not to be confused with the Video Processing Unit (also called VPU) for decoding and encoding videos. This processor is a special development for machine vision with focus on possible low power consumption. In order to achieve this, the built-in vector processors, hardware accelerators and memory architecture are optimally coordinated. Furthermore, the multi-core and multi-channel memory subsystem and the caches are software controlled, which enables a higher workload. The performance of the Myriad X VPU comes from 16 SHAVE processors, 2 RISC CPUs with a clock frequency of 700MHz and the high-performance video hardware filters. To reduce latency, data is stored on the internal 4GB DRAM. Ionica and Gregg 2015; Rivas-Gomez et al. 2018; *Intel® Neural Compute Stick 2 Product Specifications* 2020

In order for the SHAVE processors to deliver high performance with low energy, they have a wide and deep register that controls several functional units with a variable-length long instruction word (VLLIW). High parallel readability is achieved by SIMD. The SHAVE processors combine the advantages of a GPU, DPS and RISC. As with the GPU, 8-, 16-, and 32-bit integer and 16-, 32-bit floating point arithmetic can be used. Furthermore, it is easier to programming of multi-core tasks. In the figure 5.2 the block diagram of Streaming Hybrid Architecture Vectore Engine is displayed. Barry, Brick, et al. 2015

The advantage of the Intel NCS2 is that it supports integer and floating point arithmetic, same as the GPU but with a similar power consumption as the Coral. Like the GPU it is also more platform independent than the Edge TPU. But as can seen in the table, it is much slower than the other devices and also not all networks can be compiled for the NCS2.

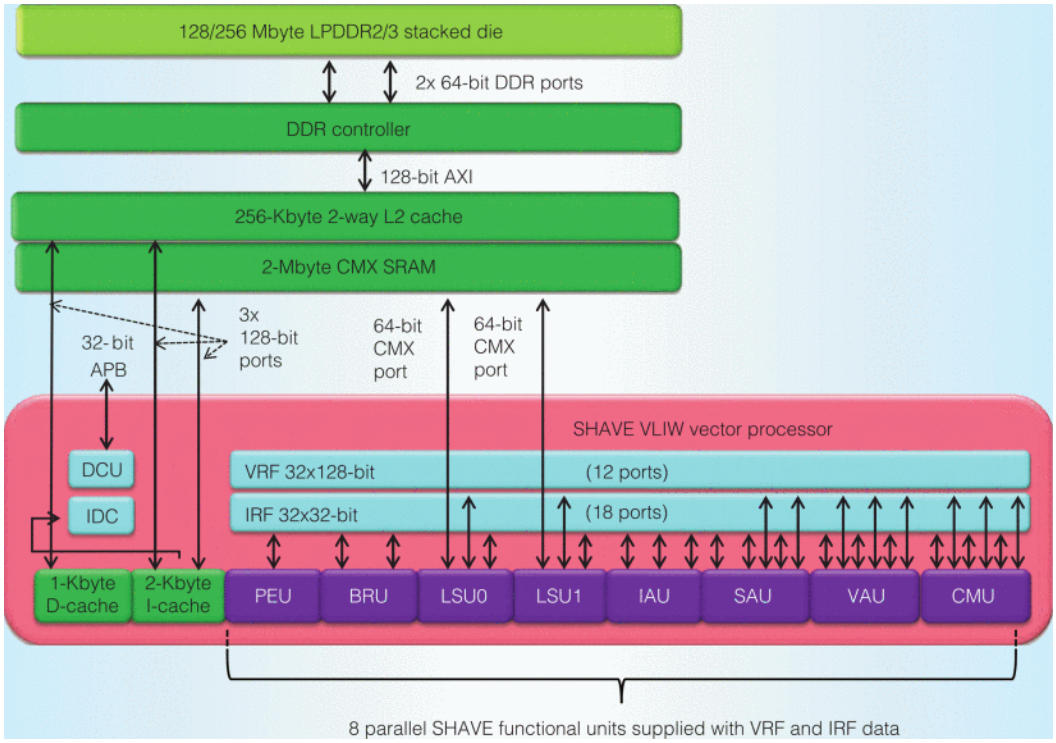


Figure 5.2: Block diagram of Streaming Hybrid Architecture Vectore Engine with the functional units PEU (predicated execution unit), BRU (branch and repeat unit), two 64-bit LSU (load and store unit), 128-bit VAU (vector arithmetic unit), 32-bit IAU (integer arithmetic unit), 32-bit SAU (skalar arithmetic unit) and a 128-bit CMU (compare and move unit)
Source: Barry, Brick, et al. 2015

Model architecture	***Desktop CPU**	****Embedded CPU**	**Desktop CPU TPU CUA**	*****Dev Board Edge TPU**
DeepLab V3 (513x513)	2 FPS	0,8 FPS	19 FPS	4 FPS
DenseNet (224x224)	2 FPS	0,9 FPS	50 FPS	40 FPS
Inception v1 (224x224)	11 FPS	2 FPS	294 FPS	243 FPS
Inception v4 (299x299)	1 FPS	0,3 FPS	11 FPS	9 FPS
Inception-ResNet V2 (299x299)	1 FPS	0,3 FPS	17 FPS	14 FPS
MobileNet v1 (224x224)	18 FPS	6 FPS	416 FPS	416 FPS
MobileNet v2 (224x224)	19 FPS	8 FPS	384 FPS	384 FPS
MobileNet v1 SSD (224x224)	9 FPS	2 FPS	153 FPS	90 FPS
MobileNet v2 SSD (224x224)	9 FPS	3 FPS	183 FPS	71 FPS
ResNet-50 V1 (299x299)	2 FPS	0,5 FPS	20 FPS	17 FPS
ResNet-50 V2 (299x299)	1 FPS	0,5 FPS	20 FPS	16 FPS
ResNet-152 V2 (299x299)	0,5 FPS	0,2 FPS	7 FPS	6 FPS
SqueezeNet (224x224)	18 FPS	4 FPS	476 FPS	500 FPS
VGG16 (224x224)	1 FPS	0,2 FPS	3 FPS	3 FPS
VGG19 (224x224)	0,9 FPS	0,1 FPS	3 FPS	3 FPS

*Desktop CPU: 64-bit Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
**Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz
***Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU

Table 5.3: Comparison different quatize model architectures with Google TPU, Desktop CPU and Embedded CPU
Source: https://coral.ai/docs/edgetpu/benchmarks/

Model Architecture	Application	Framework	Nvidia Jetson Nano	Raspberry Pi3 + NCS2	Google Edge TP Dev Board
Res-Net 50 (224x224)	Classification	Tenorflow	36 FPS	16 FPS	DNR
MobileNet-v2 (300x300)	Classification	Tensorflow	64 FPS	30 FPS	130 FPS
SSD ResNet-18 (960×544)	Object Detection	Tensorflow	5 FPS	DNR	DNR
SSD ResNet-18 (480×272)	Object Detection	Tensorflow	16 FPS	DNR	DNR
SSD ResNet-18 (300×300)	Object Detection	Tensorflow	18 FPS	DNR	DNR
SSD Mobilenet-V2 (960×544)	Object Detection	Tensorflow	8 FPS	1.8 FPS	DNR
SSD Mobilenet-V2 (480×272)	Object Detection	Tensorflow	27 FPS	7 FPS	DNR
SSD Mobilenet-V2 (300×300)	Object Detection	Tensorflow	39 FPS	11 FPS	48 FPS
Inception V4 (299×299)	Classification	PyTorch	11 FPS	DNR	9 FPS
Tiny YOLO V3 (416×416)	Object Detection	Darknet	25 FPS	DNR	DNR
OpenPose (256×256)	Pose Estimation	Caffe	14 FPS	5 FPS	DNR
VGG-19 (224×224)	Classification	MXNet	10 FPS	5 FPS	DNR
Super Resolution (481×321)	Image Processing	PyTorch	15 FPS	0.6 FPS	DNR
Unet (1x512x512)	Segmentation	Caffe	18 FPS	5 FPS	DNR

Table 5.4: Inference performance results from Jetson Nano, Raspberry Pi 3, Intel Neural Compute Stick 2, and Google Edge TPU Coral Dev Board
Source: https://devblogs.nvidia.com/jetson-nano-ai-computing/
DNR = "did not run"

6 Convolutional Neuronal Network Architecture

Convolutional Neuronal Networks are currently the first choice for network architectures in the application area of image classification, recognition and segmentation. CNN are classical feedforword networks. Which means that the information of the input image flows through the network without feedback. The general structure of a CNN consists of several convolution layers. A convolution layer contains a pair of a convolution function followed by a pooling function, sometimes also a subsampling function. The layers are all connected together to form a fully connected layer structure. Networks which have many layers are called Deep Convolutional Neuronal Network. The illustration 6.1 gives a simplified view of the structure of a CNN.Rawat and Z. Wang 2017; Albawi, Mohammed, and Al-Zawi 2017

In this thesis we deal with CNNs which are specially designed for mobile applications. These 3 models are the MobileNet-v2, the NASNet-Mobile and the EfficientNet-B0. These models are available as pre-trained models in the Tensorflow/Keras framework which is used here. These models are therefore tested for accuracy, inference speed and efficiency on the 3 available AI edge devicesKeras n.d.

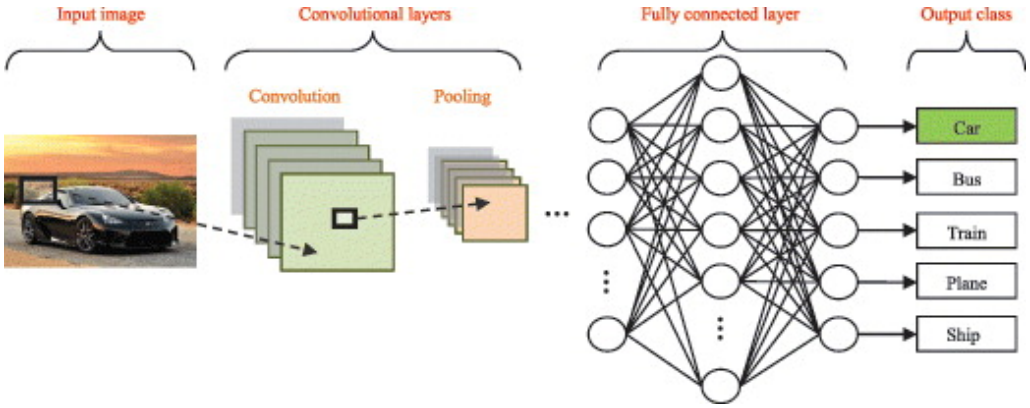
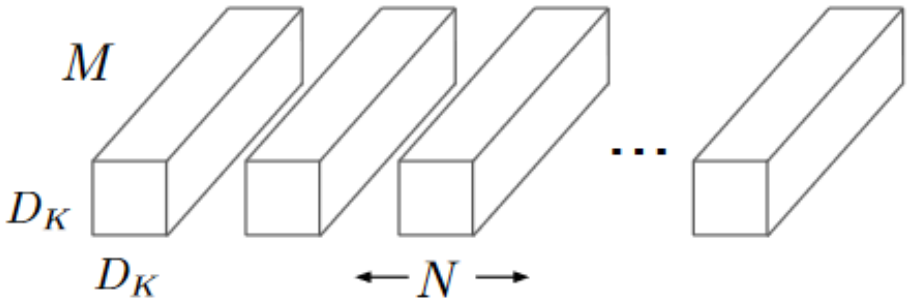


Figure 6.1: General architecture of a convolutional neuronal network
Source: Rawat and Z. Wang 2017

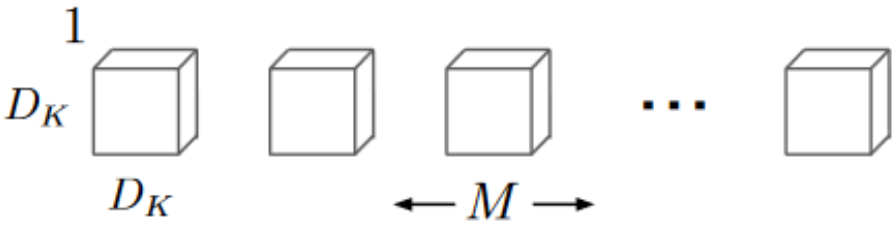
6.1 MobileNet-v2

The MobileNet-v1 is a Convolutional Neuronal Network class designed by Google which is mainly located in the mobile application area. The main focus of the MobileNet class is the latency time which was ignored in previous compact CNN. In order to achieve a higher speed the MobileNets use depthwise separable folding layers illustrate in figure 6.3a. Such a layer consists of a depthwise convolution filter and a pointwise convolution filter. This significantly reduces the costs for the calculation effort compared to the use of a standard convolution layer. With the standard convolution a 3D filter with the dimensions height, width and depth is used to determine the property from the image. For MobileNets this is divided into 2 steps. In the first step, the depth-wise convolution, a 2D filter is applied separately for each depth (channel), but no new features are determined here yet. In the second step of the pointwise convolution, the separated channels are merged again to determine the features. In the figure 6.2 you observe the difference between the standard convolution and the depthwise separable convolution. After each filter there is a batchnorm and a ReLU6 activation function, see figure 6.6. At the end of the network there is a softmax layer for classification. Howard et al. 2017

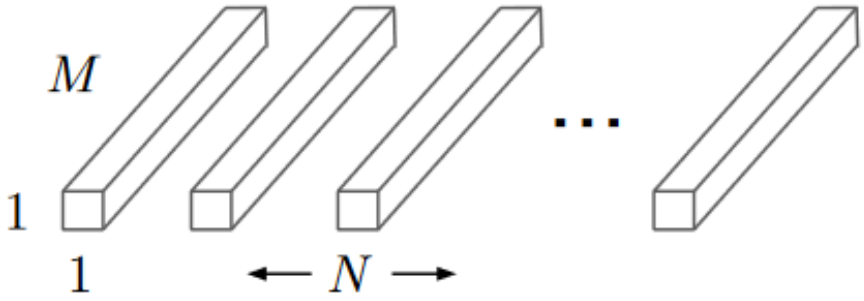
The MobileNet-v2 improves the MobileNet-v1 by replacing the depthwise separable convolution block by a bottleneck residual block, see figure 6.3b. The idea of the MobileNet-v2 is to bring the more interesting information of the input tensor into a smaller subspace. For this purpose, a 1x1 extension convolution filter and a residual connection is added to the depth-separable convolution block. The functionality of the pointwise convolution filter also changes. With MobileNet-v1 the number of channels remains the same or can be increased, with MobileNet-v2 on the other hand the number of channels is reduced, hence the name 1x1 projection convolution. Reducing the number of channels reduces the amount of data flowing through the Convolutional Neuronal Network. To avoid loss of information, the channels are expanded again with the 1x1 projection convolution at the input of the remaining bottleneck residual block 6.3b. The operation can be understood as a compress-decompress cycle. The residual connection is responsible for a better flow of the gradient, as seen in ResNet Targ, Almeida, and Lyman 2016. It is only used when the number of channels between input and output is the same, which is not often the case. Furthermore the batch standard and ReLU6 is used again as activation function as well as softmax for classification. Only after the projection convolution no activation function is used because this leads to an extensive loss of information. The structure of the MobileNet-v2 is shown in the table 6.1.Sandler et al. 2018



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

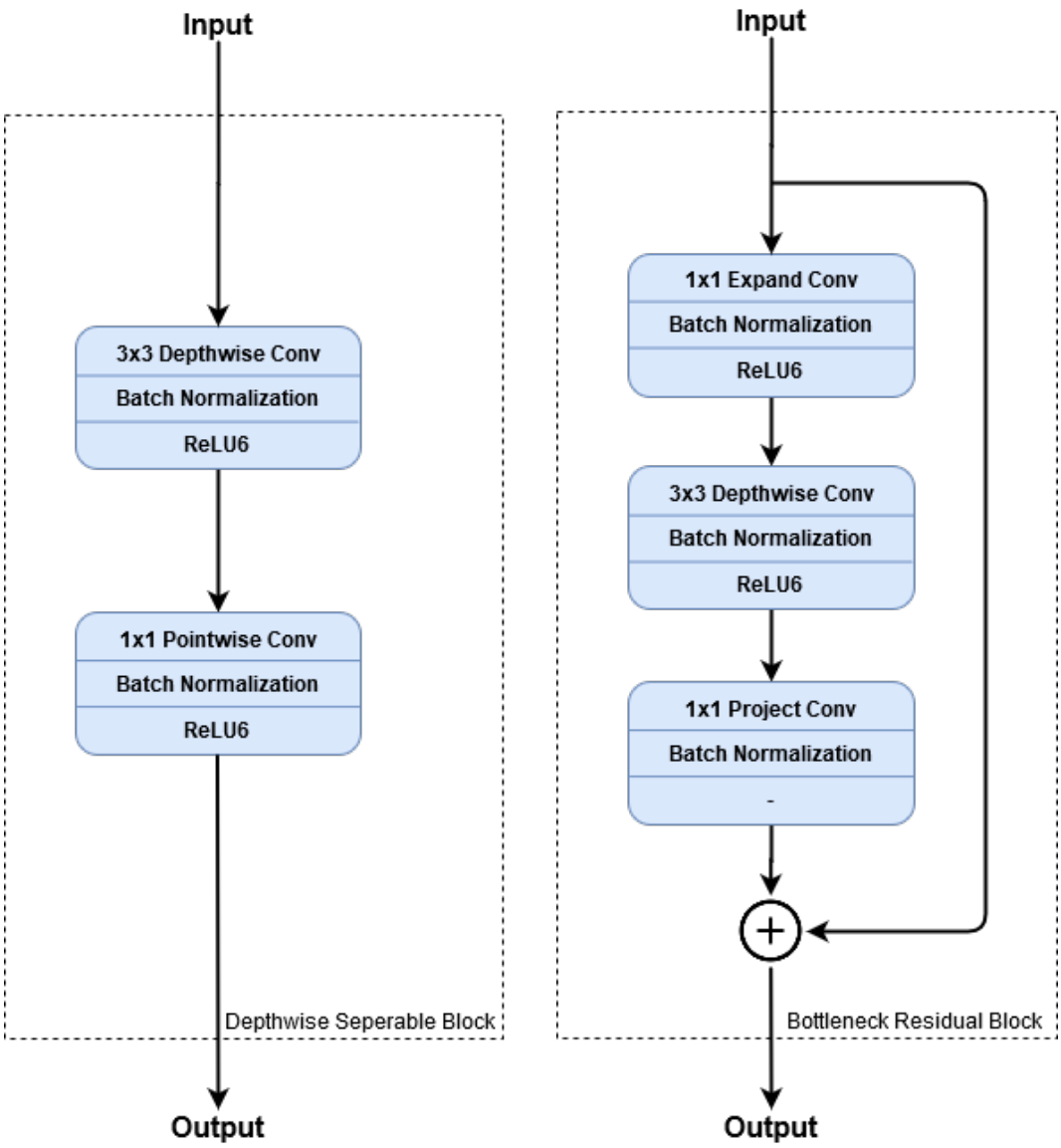


(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 6.2: The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter
Source: Howard et al. 2017

Stage i	**Operator** F_i	**Resolution** H_i **x** W_i	**Channels** C_i
1	Conv2d	224^2 x 3	32
2	Bottleneck	112^2 x 32	16
3	Bottleneck	112^2 x 16	24
4	Bottleneck	56^2 x 24	40
5	Bottleneck	28^2 x 32	80
6	Bottleneck	14^2 x 64	112
7	Bottleneck	14^2 x 96	192
8	Bottleneck	7^2 x 160	320
9	Conv2d 1x1	7^2 x 320	1280
10	Average Pooling 7x7	7^2 x 1280	-
11	Conv2d 1x1	1 x 1 x 1280	$k_{classes}$

Table 6.1: Base structure of the Mobilenet-v2
Source: Sandler et al. 2018



(a) Depthwise Seperable Convolution Block (b) Bottleneck Residual Block

Figure 6.3: Difference of the Convolution Block between Mobilenet-v1 and Mobilenet-v2

6.2 NASNetMobile

The Neural Architecture Search is the version of the MnasNet adapted for the Keras platform, which is also a Convolutional Neuronal Network developed by Google. NAS stands for Neural Architecture Search and is an automatic design technique for creating artificial neural networks. The idea of NAS is to optimize existing CNN architectures, in case of MnasNet the MobileNet-v2. The MNV2 Structure is divided into 7 blocks with a variable number of repeatable layers. For the optimization a partial search space for the blocks is defined as follows.

- Convolution operation: standard convolution, depthwise convolution, inverted bottleneck convolution
- Kernal size: 3x3 or 5x5
- Squeeze and excitation ratio: 0, 0.25
- Skip operation: poolling, identity residual or no skip
- Filter size: F_i
- Layer number: N_i

By reinforcement learning, the parameters in the partial search space are optimized for high accuracy and low latency. A recurrent neural network (RNN) based controller is used to change the parameters. The feedback for the accuracy comes from a training engine and the latency from a mobile phone engine. The architecture resulting from this optimization can be seen in the figure 6.4 Zoph and Le 2017; Zoph, Vasudevan, et al. 2018

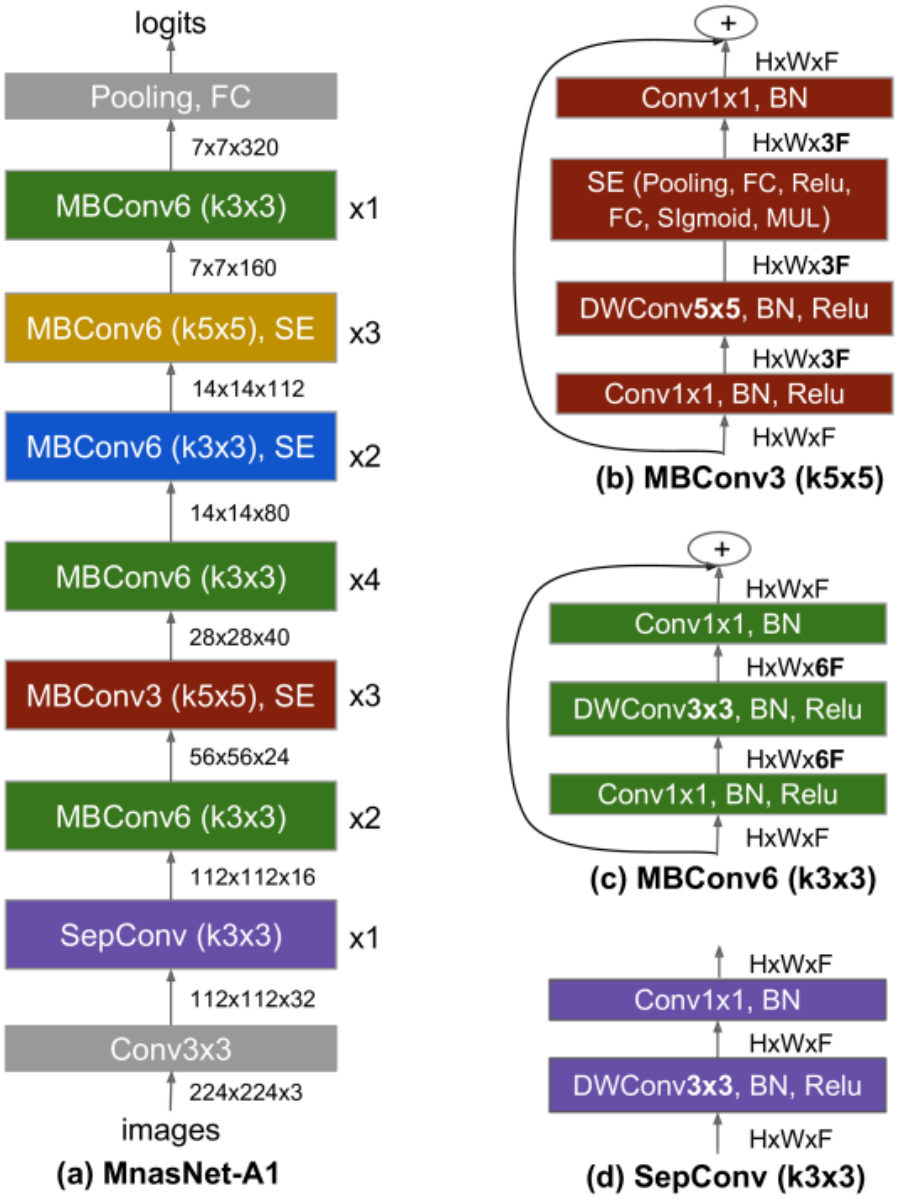


Figure 6.4: Result architecture for the mobile NAS model after optimisation and structure of the convolution blocks
Source: Tan, Chen, et al. 2019

6.3 EfficientNet-B0

The EfficientNets are another Convolutional Neuronal Network architecture from the development of Google Research. Here a new method for scaling a neural network is applied. In general, Convolutional Neuronal Networks are scaled up to achieve better accuracy but consume more resources. The most common methods are scaling up in depth or width, sometimes the resolution of the input is increased. However, only one dimension is always increased. The idea of EfficientNet is to use a compound scaling method that increases all 3 dimensions equally with a scaling constant ϕ in the equations 6.3 and general scaling methods are seen in the illustration 6.5. By setting the scaling constant the scaling can be adjusted to the available computing power. Scaling does not change the model architecture, so a good basic architecture is essential. For this purpose, a similar structure as for MnasNet is used, but the EfficientNet uses mobile inverted bottleneck convolution layer (MBConv) 6.6c with a Swish activation function, see figures 6.4,6.7. The MBConv is a combination of a standard bottleneck convolution, figure 6.6b and a squeeze and excitation block, figure 6.6a. The structure of the EfficientNet-B0 is illustrated in the table 6.2. The values α, β, γ are searched with a small grid search and for the EfficientNet-B0 the following values were found $\alpha = 1.2, \beta = 1.1, \gamma = 1.15$ under the condition that $\alpha * \beta^2 * \gamma^2 \approx 2$. Sandler et al. 2018 Tan, Chen, et al. 2019

$$depth : d = \alpha^{\phi} \tag{6.1}$$

$$width : w = \beta^{\phi} \tag{6.2}$$

$$resolution : r = \gamma^{\phi} \tag{6.3}$$

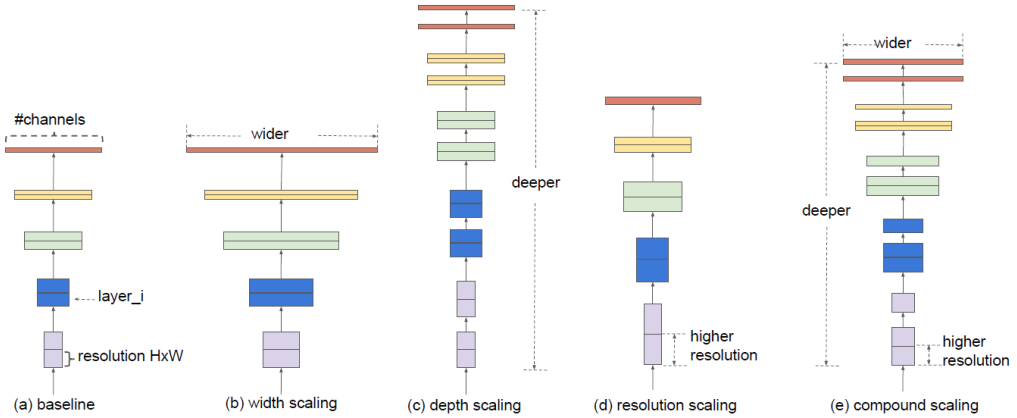
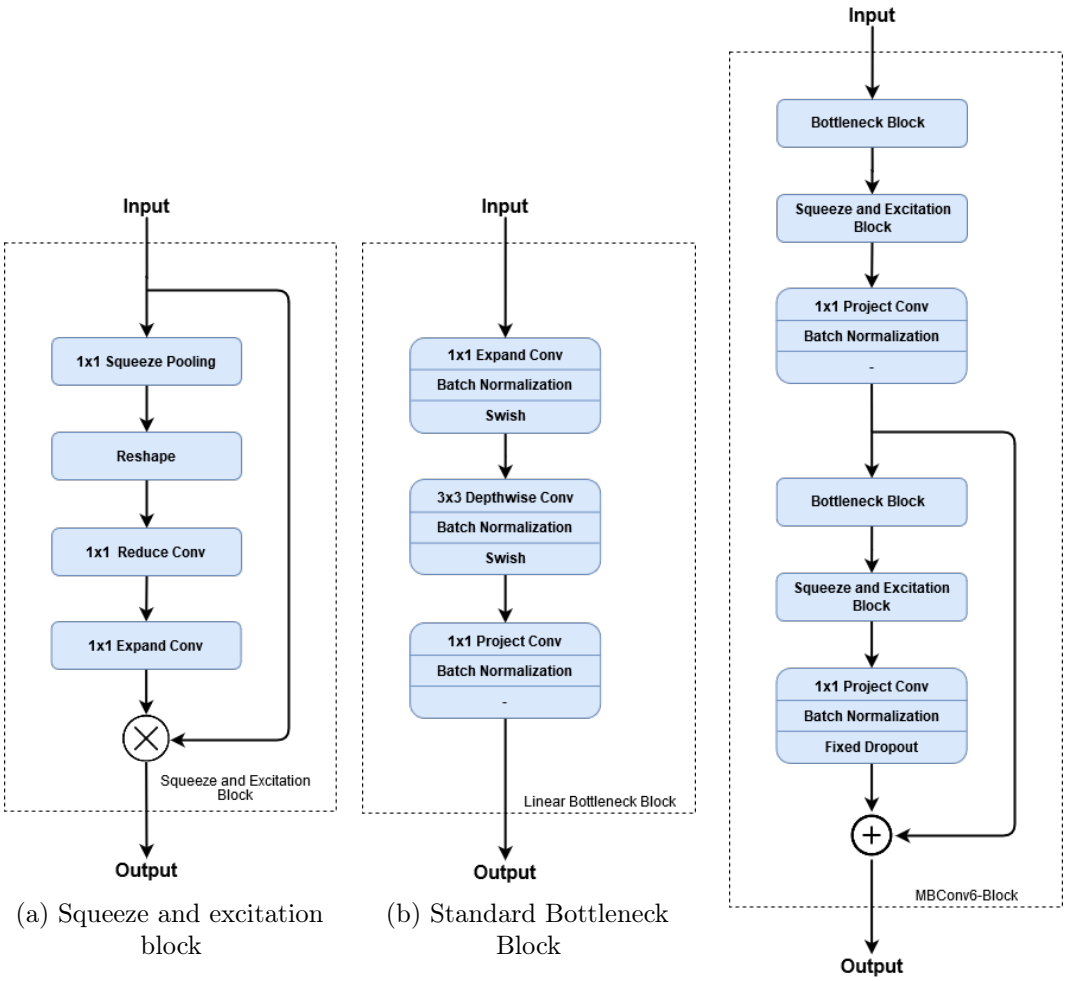


Figure 6.5: Illustration of different forms of scaling for convolution neuronal networks Source: Tan, Chen, et al. 2019



(a) Squeeze and excitation block

(b) Standard Bottleneck Block

(c) Layer structure overview of the mobile inverted Bottleneck MBConv6 layer with squeeze and excitation optimization

Figure 6.6: Layer structure overview of the squeeze convolution block and linear bottleneck
Source: Zoph, Vasudevan, et al. 2018

Stage i	**Operator** F_i	**Resolution** H_i **x** W_i	**Channels** C_i
1	Conv3x3	224 x 224	32
2	MBConv1, k3x3	112 x 112	16
3	MBConv6, k3x3	112 x 112	24
4	MBConv6, k5x5	56 x 56	40
5	MBConv6, k3x3	28 x 28	80
6	MBConv6, k5x5	14 x 14	112
7	MBConv6, k5x5	14 x 14	192
8	MBConv6, k3x3	7 x 7	320
9	Conv1x1 & Pooling & FC	7 x 7	1280

Table 6.2: Base structure of the EfficientNet-B0
Source: Tan and Le 2019

Swish function:
$$swish(x) = x * \sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \tag{6.4}$$

β constant or trainable parameter

Sigmoid function:
$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.5}$$

Rectified linear unit:
$$ReLU(x) = max(0, x) = \left\{ \begin{array}{l} 0, if x < 0 \\ x, if x \geq 0 \end{array} \right\} \tag{6.6}$$

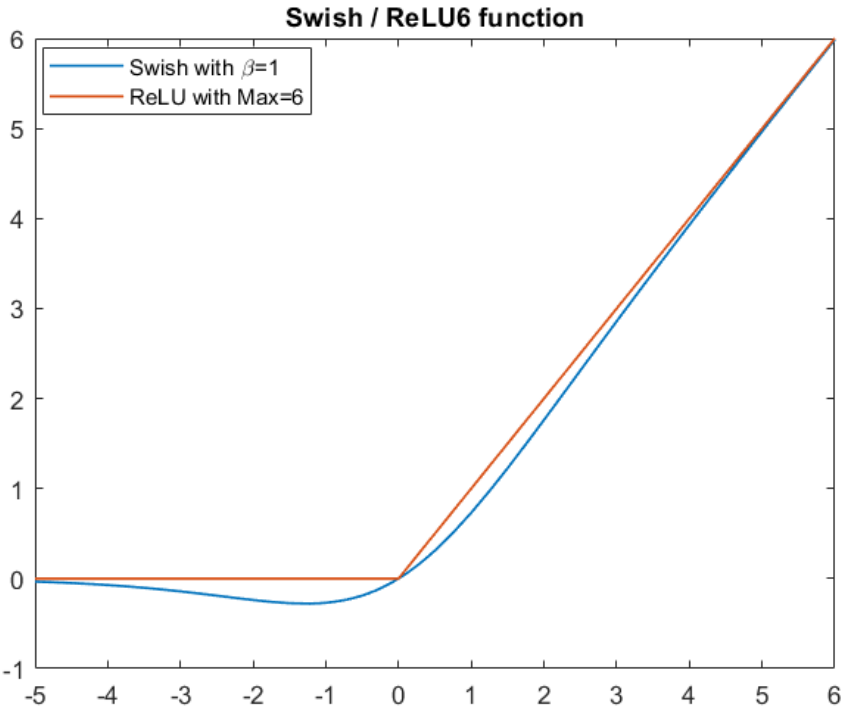


Figure 6.7: Illustration of the Swish in comparison to the ReLu6 activation function
Source: Ramachandran, Zoph, and Le 2017

7 Training of the Neuronal Networks

The retraining of the base models MobileNet-v2, NASNet-Mobile and EfficientNet-B0 takes places on Colab. This has the benifit to use Google hardware accelerators, what will reduce the time of training distictly, as the own hardware usually is not as capable. Google provides two hardware accelerators, a cloud GPU and a TPU. In this work both processing units are used to create a model. It will be observed if there is an impact of a TPU trained model on the Edge-TPU.

For the problem in this work the output of the neuronal network is set to 11 classes, see figure 7.1. One class for the background and 10 classes for defined waste categories.

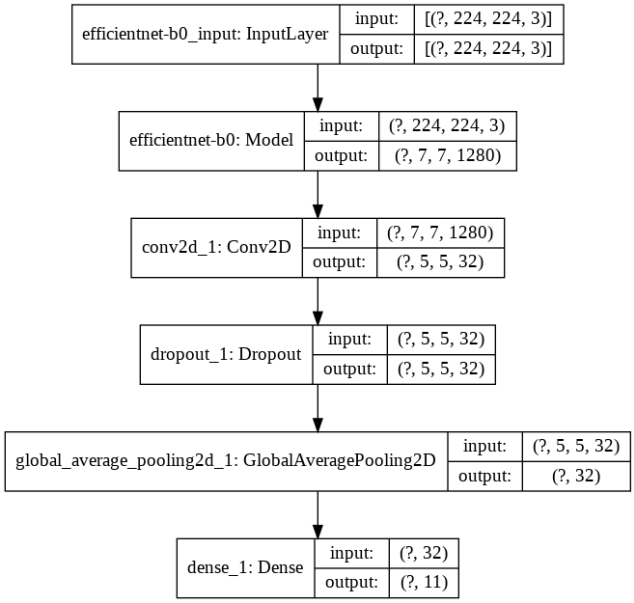


Figure 7.1: EffNet example of the added output layers to brake down the output from 1000 classes to 11. The input is defined for a 3 channel image with a 224x224 resolution. The '?' stands for the batchsize which will be defined at the inference

7.1 Parameters

Tensorflow 2.1.0 is used for training the CNNs, which is the latest stable version at the time of training.

The dataset contains 25,681 images from 11 classes and is splited in 66% training, 22% validiation and 11% test samples. The dataset where converted into TFrecords file to get a efficient way to train with Tensorflow.

For the TPU training the TFrecords files has to be stored in a Google Cloude Storage Bucket otherwise it is not possible to cache the dataset, see under "Cannot use local filesystem" at *Troubleshooting | Cloud TPU* 2020.
For the GPU both system Google drive and Cloud Storage are possible and this case Google drive is used.

The Tensor Processing Unit only supports tf.float32, tf.int32, tf.bfloat16 and tf.bool, see under "Unsupported data typ" *Troubleshooting | Cloud TPU* 2020. Wherefore the input image is cast to tf.float32 and input label to tf.int32.
The GPU can handle any tf.datatypes so the input is set to tf.float16 and the ouput to uint8. With the GPU it is therefore possible to adapt the Keras models better to the specifications of the edge hardware than is possible with the TPU .

Both models get compiled with the "Adam-optimizer" (Kingma and Ba 2017) and "categorical cross-entropy", see equation 7.2. The result of this loss function is a measure of the quality of the probability from the Softmax activation function, seen at equation 7.1. These parameters are often used for models with One-Hot-Encoding. In both cases the batchsize is set on 64 samples and 20 epochs for the full training. The number of steps is calculated with the max number of training samples divided by the batchsize for GPU-accelerator.
TPUs are design to handle large batchsizese. To get the best perfromance the batchsize is multiplied by the number of cores, in this case are 8 cores available. In one step 8 cores calculate parallel the gradient from 64 samples, therefore the steps per epoch decrease by the number of cores. Anyway to have the same quantity of steps per epoch and consequently a similar result, the batchsize is not multiplied by the number of cores. The table 7.1 gives an overview of the use training settings with the cloud GPU and the cloud TPU.

The table 7.2 show the trainable and total parameters of each base Keras model. As a rule of thumb, more parameters mean a better accuracy and slower inference. A larger number of parameters means in any case a greater training effort. It can be seen that the MNV2 has fewer parameters than the other two models. Therefore the MNV2 is faster to train and should have higher inference speed than the NAS-M and EffNet which has nearly the same amout of parameters.

Softmax function:
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{7.1}$$

Cross-Entropy function:
$$CE = -\frac{1}{K} \sum_{j}^{C} l_j * log(\sigma(z)_j) \tag{7.2}$$

K dimension of output tensor
z input vector with K dimensions
l ground truth vector, one-hot encoded label of the true class

Setting	**GPU-Accelerator**	**TPU-Accelerator**
TF.Version	TF 2.1.0	TF 2.1.0
Train samples	17916	17916
Valid. samples	5098	5098
Dataset fromat	TFrecord	TFrecord
Storage	Google Drive	Google Cloud Storage
Input shape	(None,224,224,3)	(None,224,224,3)
Input output	(None,16)	(None,16)
Types image	tf.float16	tf.float32
Types label	tf.uint8	tf.int32
Encoding	One-Hot-Encoding	One-Hot-Encoding
Optimizer	Adam optimizer	Adam optimizer
Loss function	Categorical crosstropy	Categorical crosstropy
Batch size	64	64
Epochs	20	20
Steps	280	279
Learning rate schedular	epoch <= 20 = 1e-4 20 >epoch <=40 = 1e-5 40 >epoche <=50 = 1e-6	epoch <= 20 = 1e-4 20 >epoch <=40 = 1e-5 40 >epoche <=50 = 1e-6

Table 7.1: Overview of learning parameters with the TPU and GPU accelerators

Parameters	MobileNet-v2	NASNet-Mobile	EfficientNet-B0
Trainable Parameters	2,592,907	4,537,501	4,376,583
Non-trainable Parameters	34,112	36,738	42,016
Total Parameters	2,627,019	4,574,239	4,418,599

Table 7.2: Overview of learning parameters with the TPU and GPU accelerators

7.2 Training time

In this section we take a look on the training velocity of the different models on the two cloud accelerators. The time to train one model depends on the model structure, dataset size and training settings. The time, that is needed to update the weights of neurons, is a indicator of the cost to create a new model.

An epoch stands for a complete run of the data set. The step is an update of the gradient, whereby it is usual to use the batch size to calculate the number of steps with $Epoch = \frac{Dataset\_size}{Batch\_size}$. Furthermore, the total time for an epoch and a step is determined, as well as the error and the accuracy during training as during the validation.

The table 7.2 shows the number of parameters to be trained. It follows that the MNV2 model takes the least time to recalculate the weights. Followed by the EffNet and the NAS-M takes the most time. A look at the following 3 illustrations 7.1, 7.3, 7.5 of the first 5 steps of each training of the models confirm the expected behavior. It is noticeable that the EffNet takes much less time compared to the 7.3, although both have similar parameters to be trained.

Compared to the TPU, the GPU needs much less time to initialize. After the initialization the TPU has a clear advantage over the GPU. Training time with the TPU is 66% less with the MobileNet-v2 than training with the GPU. The Nasnet achieves 50% and the EfficientNet-B0 even 75%. For more information about the benchmark of Google's Cloud processing units see N. P. Jouppi et al. 2017 and Y. E. Wang, Wei, and Brooks 2019

When training with the TPU, all 8 cores are always used to update the weights. In contrast to the GPU where only the required number of cores is used. To get the best performance out of the TPU, each core should be used at full capacity. Usually the $Batch\_size$ is simply multiplied by 8. To get a better comparison, this was not done in this training.

MobileNet-v2

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
280/280 [==============================] - 91s 325ms/step - loss: 0.8515 - accuracy:
    0.7520 - val_loss: 3.3345 - val_accuracy: 0.4469 - lr: 1.0000e-04

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
280/280 [==============================] - 85s 302ms/step - loss: 0.6241 - accuracy:
    0.8188 - val_loss: 2.9255 - val_accuracy: 0.5277 - lr: 1.0000e-04

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
280/280 [==============================] - 85s 305ms/step - loss: 0.3644 - accuracy:
    0.8938 - val_loss: 2.6677 - val_accuracy: 0.6160 - lr: 1.0000e-04

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
280/280 [==============================] - 85s 304ms/step - loss: 0.2160 - accuracy:
    0.9381 - val_loss: 1.6893 - val_accuracy: 0.7318 - lr: 1.0000e-04

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
280/280 [==============================] - 85s 303ms/step - loss: 0.1532 - accuracy:
    0.9546 - val_loss: 1.3364 - val_accuracy: 0.7949 - lr: 1.0000e-04
```

Listing 7.1: First 5 epochs of pretraining the Mobilenet-v2 with GPU accelerator

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
279/279 [==============================] - 256s 919ms/step - loss: 1.8706 - accuracy:
    0.4010 - val_loss: 2.4619 - val_accuracy: 0.4012

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
279/279 [==============================] - 24s 85ms/step - loss: 0.8831 - accuracy:
    0.7101 - val_loss: 1.2624 - val_accuracy: 0.7065

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
279/279 [==============================] - 24s 85ms/step - loss: 0.3885 - accuracy:
    0.8784 - val_loss: 0.6069 - val_accuracy: 0.8636

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
279/279 [==============================] - 24s 85ms/step - loss: 0.1435 - accuracy:
    0.9584 - val_loss: 0.2824 - val_accuracy: 0.9332

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
279/279 [==============================] - 24s 88ms/step - loss: 0.0427 - accuracy:
    0.9885 - val_loss: 0.1511 - val_accuracy: 0.9666
```

Listing 7.2: First 5 epochs of pretraining the Mobilenet-v2 with TPU accelerator

NASNet-Mobile

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
280/280 [==============================] - 225s 804ms/step - loss: 0.9086 - accuracy:
    0.7316 - val_loss: 3.5660 - val_accuracy: 0.0822

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
280/280 [==============================] - 166s 593ms/step - loss: 0.5332 - accuracy:
    0.8429 - val_loss: 3.8819 - val_accuracy: 0.2713

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
280/280 [==============================] - 166s 593ms/step - loss: 0.2630 - accuracy:
    0.9231 - val_loss: 4.1208 - val_accuracy: 0.3721

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
280/280 [==============================] - 167s 595ms/step - loss: 0.1413 - accuracy:
    0.9572 - val_loss: 3.3297 - val_accuracy: 0.4672

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
280/280 [==============================] - 166s 593ms/step - loss: 0.0723 - accuracy:
    0.9783 - val_loss: 2.3847 - val_accuracy: 0.5857
```

Listing 7.3: First 5 epochs of pretraining the NASNetMobile with GPU accelerator

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
279/279 [==============================] - 445s 2s/step - loss: 1.8604 - accuracy:
    0.3893 - val_loss: 2.7718 - val_accuracy: 0.1559

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
279/279 [==============================] - 78s 281ms/step - loss: 0.6299 - accuracy:
    0.8072 - val_loss: 2.8165 - val_accuracy: 0.2969

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
279/279 [==============================] - 77s 277ms/step - loss: 0.2104 - accuracy:
    0.9380 - val_loss: 1.9313 - val_accuracy: 0.5192

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
279/279 [==============================] - 78s 279ms/step - loss: 0.0549 - accuracy:
    0.9847 - val_loss: 1.2406 - val_accuracy: 0.6799

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
279/279 [==============================] - 78s 278ms/step - loss: 0.0190 - accuracy:
    0.9953 - val_loss: 0.8056 - val_accuracy: 0.7890
```

Listing 7.4: First 5 epochs of pretraining the NASNetMobile with TPU accelerator

EfficientNet-B0

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
280/280 [==============================] - 149s 531ms/step - loss: 1.0536 - accuracy:
    0.6824 - val_loss: 1.8567 - val_accuracy: 0.5404

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
280/280 [==============================] - 109s 391ms/step - loss: 0.4824 - accuracy:
    0.8519 - val_loss: 1.2278 - val_accuracy: 0.6471

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
280/280 [==============================] - 109s 390ms/step - loss: 0.2623 - accuracy:
    0.9253 - val_loss: 0.4140 - val_accuracy: 0.8559

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
280/280 [==============================] - 109s 391ms/step - loss: 0.1399 - accuracy:
    0.9606 - val_loss: 0.2538 - val_accuracy: 0.9174

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
280/280 [==============================] - 108s 385ms/step - loss: 0.0825 - accuracy:
    0.9787 - val_loss: 0.1588 - val_accuracy: 0.9508
```

Listing 7.5: First 5 epochs of pretraining the Efficientnet-B0 with GPU accelerator

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 1/20
279/279 [==============================] - 141s 507ms/step - loss: 1.6444 - accuracy:
    0.4912 - val_loss: 0.4626 - val_accuracy: 0.8899

Epoch 00002: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 2/20
279/279 [==============================] - 28s 101ms/step - loss: 0.4648 - accuracy:
    0.8652 - val_loss: 0.1683 - val_accuracy: 0.9487

Epoch 00003: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 3/20
279/279 [==============================] - 28s 99ms/step - loss: 0.1413 - accuracy:
    0.9638 - val_loss: 0.0842 - val_accuracy: 0.9781

Epoch 00004: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 4/20
279/279 [==============================] - 29s 104ms/step - loss: 0.0579 - accuracy:
    0.9861 - val_loss: 0.0737 - val_accuracy: 0.9802

Epoch 00005: LearningRateScheduler reducing learning rate to 0.0001.
Epoch 5/20
279/279 [==============================] - 29s 105ms/step - loss: 0.0292 - accuracy:
    0.9940 - val_loss: 0.0646 - val_accuracy: 0.9838
```

Listing 7.6: First 5 epochs of pretraining the Efficientnet-B0 with TPU accelerator

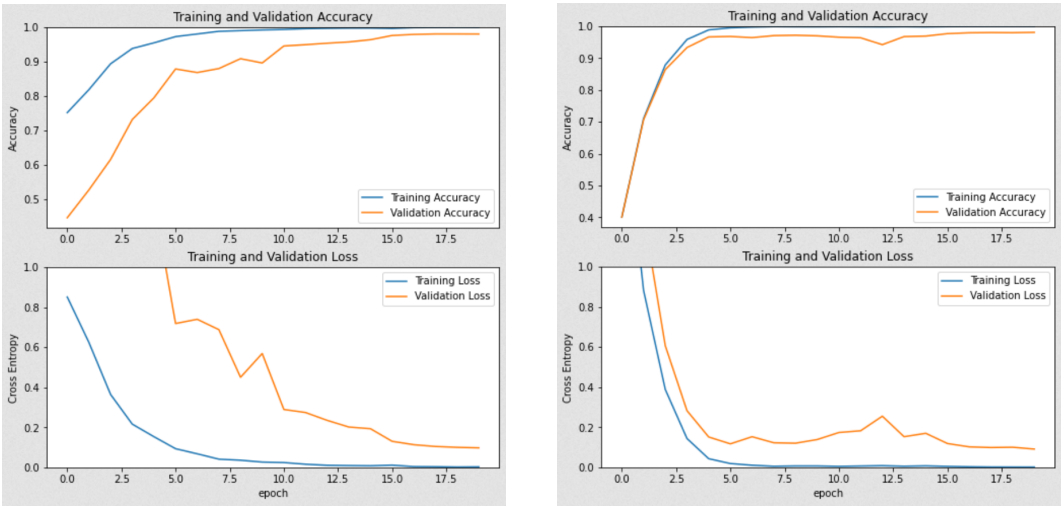
7.3 Training evalutation

In this section the learning curves of the training of the respective models are evaluated. The diagrams show the course of the determined accuracy (definition is in chapter 7.4) and the loss of the cross-entropy 7.2 during training and validation.

The best result was achieved with EfficientNet-B0 with an accuracy value of >0.98 and a loss value < 0.1. Also with MobileNet-v2 and NASNet-Mobile an acceptable result was achieved with >0.98 accuracy and < 0.2 loss.
It is noticeable that the validation values at the beginning of the TPU training of the EfficientNet-B0 are better than the training values. This could be due to the fact that the validation takes place after an epoch and the curves rise and fall very quickly.

A smoother training process can be noticed with the TPU models, which have significantly less deflections than the training processes of the GPU models. Optimal values are also reached faster with the TPU than with the GPU.

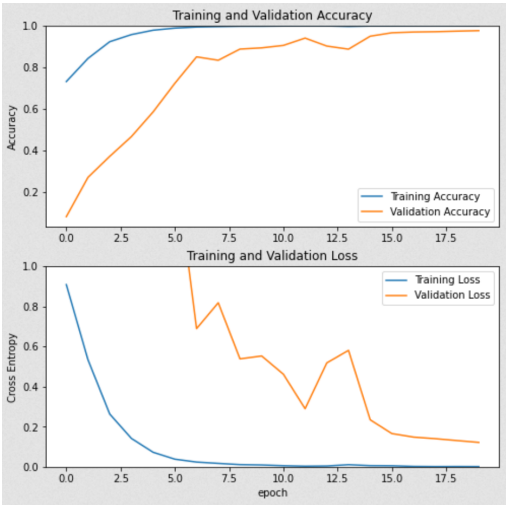
MobileNet-v2



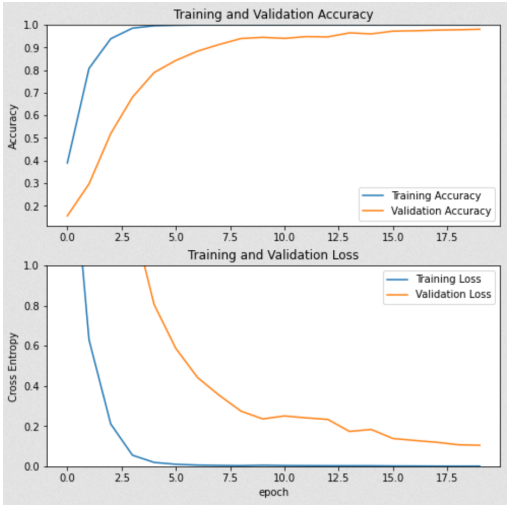
(a) with GPU-Accelerator (b) with TPU-Accelerator

Figure 7.2: MobileNet-v2 training and validation accuracy and loss result

NASNet-Mobile



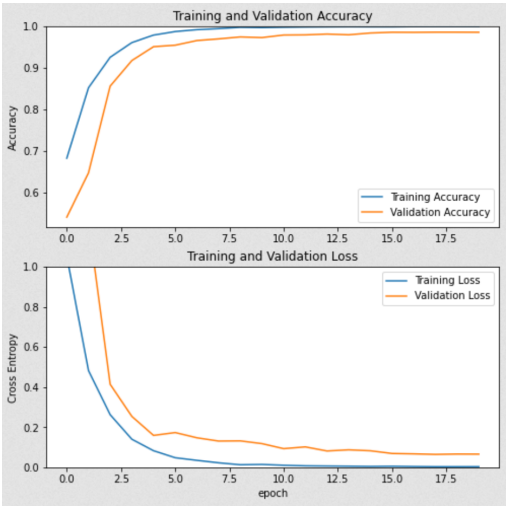
(a) with GPU-Accelerator



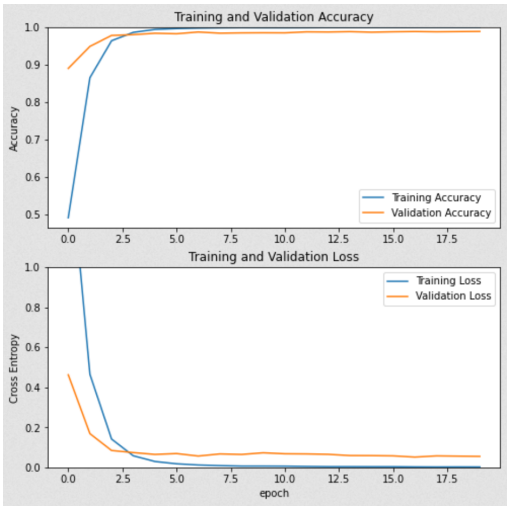
(b) with TPU-Accelerator

Figure 7.3: NASNet-Mobile training and validation accuracy and loss result

EfficientNet-B0



(a) with GPU-Accelerator



(b) with TPU-Accelerator

Figure 7.4: EfficientNet-B0 training and validation accuracy and loss result

7.4 Confusion Matrix of Keras models

In this section the accuracy, seen in the equation 7.3 of the models is determined with 2667 test images. A confusion matrix is created for each model. The columns represent the correct class and the rows represent the predicted class.

The confusion matrices reflect the results of the learning curves. With an overall accuracy of 0.954 and an macro-F1 (see chapter 7.5) score of 0.951, the GPU-trained EfficientNet-B0, see matrix 7.7a scores best. The worst prediction is obtained by NASNet-Mobile with TPU training seen at matrix 7.6b with an accuracy of 0.8943 and macro-F1 of 0.879. The accuracy (equation: 7.3) and the macro F1-score (equation: 7.7) are general indicators of the quality of CNN. The accuracy indicates the ratio between all correct (True Positive + True Negative) to all predicted samples to the complete number of samples (True Positive + True Negative + False Positive + False Negative). The macro F1-score instead use the harmonic mean of precision and recall what is more accurate for a unbalanced matrix.

It is noticeable that TPU trained models perform on average 2 % worse than GPU models. TPUs are not as flexible as GPU, which we can see in the supported arithmetic which affects the accuracy accordingly. Furthermore the TPU is designed for larger data sets than used in this project.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7.3}$$

MobileNet-v2

Keras MobileNet-v2 GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	63	1	2	0	0	4	0	1	0	3	0
food_bowl	1	157	0	0	0	0	0	1	0	0	0
food_box	4	2	215	2	0	1	0	1	0	1	0
glas_bottle	0	0	0	216	11	0	4	1	0	0	0
metal_can	0	0	0	9	239	0	0	0	0	0	0
plastic_bag	0	0	1	3	0	238	0	1	0	0	0
plastic_bottle	0	3	2	38	1	1	387	15	0	0	8
plastic_cup	1	15	0	2	0	0	4	340	1	0	0
plastic_cutlery	2	4	3	3	0	1	1	0	244	0	0
snack_wrap	0	0	2	0	0	0	0	0	0	218	0
tetrapack	0	0	0	4	0	0	0	0	0	2	183

right	2500
wrong	167
accuracy	0.937
macro-F1	0.936

(a) trained with GPU-Accelerator

Keras MobileNet-v2 TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	66	5	3	4	2	4	1	6	2	2	2
food_bowl	1	167	2	0	0	0	0	9	2	0	0
food_box	2	0	211	0	0	0	0	0	1	1	0
glas_bottle	0	0	0	177	2	0	14	2	0	0	1
metal_can	0	2	0	10	242	0	1	1	0	0	3
plastic_bag	1	1	3	4	0	239	0	8	0	1	0
plastic_bottle	1	3	0	48	0	1	375	9	1	1	2
plastic_cup	0	3	0	4	0	0	0	325	0	0	0
plastic_cutlery	0	1	2	1	0	0	1	0	239	0	0
snack_wrap	0	0	4	5	1	1	0	0	0	209	1
tetrapack	0	0	0	24	4	0	4	0	0	10	182

right	2432
wrong	235
accuracy	0.912
macro-F1	0.904

(b) trained with TPU-Accelerator

Figure 7.5: Confustion matrix from different CNN-Models on Coral USB Accelerator

NASNet-Mobile

Keras NASNet-Mobile GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	63	0	0	1	0	2	0	1	1	1	1
food_bowl	2	170	1	1	0	0	0	2	1	0	0
food_box	1	1	218	0	0	0	0	0	0	2	0
glas_bottle	2	0	0	197	10	0	19	0	0	0	0
metal_can	0	0	0	12	239	0	0	0	0	0	0
plastic_bag	0	0	1	0	0	238	0	1	0	0	0
plastic_bottle	1	1	2	60	1	2	354	22	0	1	5
plastic_cup	2	9	1	6	0	2	19	334	1	1	0
plastic_cutlery	0	0	1	0	0	0	1	0	242	1	0
snack_wrap	0	1	1	0	0	1	0	0	0	204	2
tetrapack	0	0	0	0	1	0	3	0	0	14	183

right	2442
wrong	225
accuracy	0.916
macro-F1	0.935

(a) trained with GPU-Accelerator

Keras NASNet-Mobile TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	68	2	3	3	0	0	1	0	5	1	0
food_bowl	0	170	0	0	0	0	1	3	8	0	0
food_box	0	1	210	0	0	0	0	0	1	0	0
glas_bottle	0	1	0	161	1	0	30	0	0	0	1
metal_can	1	3	1	27	242	0	3	4	1	3	4
plastic_bag	0	0	3	0	0	241	0	3	3	1	0
plastic_bottle	0	0	0	63	1	0	347	4	7	0	15
plastic_cup	0	5	2	10	1	4	14	346	2	0	0
plastic_cutlery	0	0	0	1	0	0	0	0	218	1	0
snack_wrap	2	0	6	0	0	0	0	0	0	215	4
tetrapack	0	0	0	12	6	0	0	0	0	3	167

right	2385
wrong	282
accuracy	0.894
macro-F1	0.897

(b) trained with TPU-Accelerator

Figure 7.6: Confusion matrix of the NASNet-Mobile Keras classification models

EfficientNet-B0

Keras EfficientNet-B0 GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	64	3	1	1	0	1	1	0	0	0	0
food_bowl	0	170	0	0	0	0	0	1	0	0	0
food_box	4	1	221	0	0	1	0	2	1	1	0
glas_bottle	0	0	0	210	3	0	2	0	0	0	0
metal_can	0	0	0	17	245	0	1	0	0	0	0
plastic_bag	0	0	0	2	0	241	0	2	1	1	0
plastic_bottle	2	1	1	41	1	0	390	8	0	0	1
plastic_cup	1	6	0	4	1	2	1	347	0	0	0
plastic_cutlery	0	1	0	1	0	0	0	0	243	0	0
snack_wrap	0	0	2	0	1	0	1	0	0	222	4
tetrapack	0	0	0	1	0	0	0	0	0	0	186

right	2539
wrong	123
accuracy	0.954
macro-F1	0.951

(a) trained with GPU-Accelerator

Keras EfficientNet-B0 TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	2	3	5	1	1	2	2	1	1	1
food_bowl	0	169	1	0	0	0	0	3	0	0	0
food_box	2	1	216	0	0	0	0	0	0	0	0
glas_bottle	0	0	0	190	2	0	8	0	0	0	0
metal_can	0	1	0	11	248	0	0	1	0	0	2
plastic_bag	1	0	1	4	0	243	0	1	0	1	0
plastic_bottle	0	3	3	54	0	1	386	12	0	0	0
plastic_cup	0	5	0	2	0	0	0	340	1	0	0
plastic_cutlery	0	1	1	3	0	0	0	0	243	0	0
snack_wrap	1	0	0	1	0	0	0	1	0	222	1
tetrapack	0	0	0	7	0	0	0	0	0	0	187

right	2511
wrong	156
accuracy	0.942
macro-F1	0.939

(b) trained with TPU-Accelerator

Figure 7.7: Confusion matrix of the EfficientNet-B0 Keras classification models

7.5 Precision, Recall and F1-score

This section deals with the indicators precision, recall and F1-score. These values give information about the performance of the neural network over the individual categories. Shung 2020

The precision (equation: 7.4) is the ratio of all correctly predicted units TP (True Positives) by the amount of times this class was determined as a result TP+FP (True Positives + False Positives). It thus indicates how often a class is interpreted for another class. The table 7.8a shows the precision values for each category. The 2 categories background and plastic bottle. The background class is used to determine whether an object is in front of the camera or not. A low precision value means that the object cannot be recognized as such. This has a negative effect on the performance but does not affect the quality of the sorting. In contrast to the plastic bottle class. Here it shows that some objects are wrongly recognized as plastic bottles, which has a negative effect on the quality of the sorting. In this case it is mainly glass bottles that are misinterpreted, this can be seen in the confusions matrix 7.4.

The recall (equation: 7.5) describes the ratio of all correctly classified units TP (True Positives) of a class to the total number of units TP + TN (True Positives + False Negatives). It is a measure of how well the CNN can classify a class with the given features of the class. In the table 7.8b the values of the recall are listed for the respective categories. Here the glass bottle class stands out, which shows significant low values for all models. This is due to the similar features of a glass bottle with a plastic bottle, as already mentioned for precision. Furthermore you can see that TPU trained models are better able to assign the background than GPU trained models.

The F1-score (equation: 7.6) is the harmonic average of precision and recall. In contrast to the accuracy, high values are penalized when calculating the F1-score. The F1 therefore describes neural networks with an unbalanced confusion matrix better than the accuracy of a class. The macro-F1 (equation: 7.7) is the average of all F1-scores of the Convolutional Neuronal Network and describes the quality like the general accuracy. In this case, the test dataset is relatively balanced in terms of the number of samples per class. Therefore, only a minimal difference between the accuracy and the macro F1-score. Shmueli 2020

$$Precision = \frac{TP}{TP + FP} \tag{7.4}$$

$$Recall = \frac{TP}{TP + FN} \tag{7.5}$$

TP True Positive
FP False Positive
FP False Negative

$$F1 - score = \frac{2 \cdot Recall \cdot Precision}{Recall + Precision} \tag{7.6}$$

$$MacroF1 = \frac{1}{N} \sum_{i=0}^{N} F1score_i \tag{7.7}$$

N Number of F1-scores

Precision						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.851	0.680	0.900	0.819	0.901	0.779
food_bowl	0.987	0.923	0.960	0.934	0.994	0.977
food_box	0.951	0.981	0.982	0.991	0.957	0.986
glas_bottle	0.931	0.903	0.864	0.830	0.977	0.950
metal_can	0.964	0.934	0.952	0.837	0.932	0.943
plastic_bag	0.979	0.930	0.992	0.960	0.976	0.968
plastic_bottle	0.851	0.850	0.788	0.794	0.876	0.841
plastic_cup	0.937	0.979	0.891	0.901	0.959	0.977
plastic_cutlery	0.946	0.980	0.988	0.991	0.992	0.980
snack_wrap	0.991	0.946	0.976	0.947	0.965	0.982
tetrapack	0.968	0.813	0.910	0.888	0.995	0.964

(a) Precision matrix of the Keras models trained with GPU and TPU accelerator

Recall						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.887	0.930	0.887	0.958	0.901	0.944
food_bowl	0.863	0.918	0.934	0.934	0.934	0.929
food_box	0.956	0.938	0.969	0.933	0.982	0.960
glas_bottle	0.780	0.639	0.711	0.581	0.758	0.686
metal_can	0.952	0.964	0.952	0.964	0.976	0.988
plastic_bag	0.971	0.976	0.971	0.984	0.984	0.992
plastic_bottle	0.977	0.947	0.894	0.876	0.985	0.975
plastic_cup	0.944	0.903	0.928	0.961	0.964	0.944
plastic_cutlery	0.996	0.976	0.988	0.890	0.992	0.992
snack_wrap	0.973	0.933	0.911	0.960	0.991	0.991
tetrapack	0.958	0.953	0.958	0.874	0.974	0.979

(b) Recall matrix of the Keras models trained with GPU and TPU accelerator

Figure 7.8: Precision and Recall matrix of the Keras models trained with GPU and TPU accelerator

F1-value						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.869	0.786	0.894	0.883	0.901	0.854
food_bowl	0.921	0.920	0.963	0.934	0.963	0.952
food_box	0.953	0.959	0.963	0.961	0.969	0.973
glas_bottle	0.849	0.748	0.823	0.684	0.854	0.797
metal_can	0.958	0.949	0.942	0.896	0.953	0.965
plastic_bag	0.975	0.952	0.974	0.972	0.980	0.980
plastic_bottle	0.910	0.896	0.885	0.833	0.927	0.903
plastic_cup	0.941	0.939	0.943	0.930	0.961	0.960
plastic_cutlery	0.970	0.978	0.990	0.938	0.992	0.986
snack_wrap	0.982	0.939	0.937	0.953	0.978	0.987
tetrapack	0.963	0.877	0.976	0.881	0.984	0.971
macro-F1	0.936	0.904	0.935	0.897	0.951	0.939

Figure 7.9: F1 value matrix of the Keras models trained with GPU and TPU accelerator

8 Compiling the Keras models for the edge frameworks

This chapter describes how to adapt and compile the created Keras models to the respective frameworks of the Artificial Intelligence edge devices. An overview of how the Keras model is compiled on the respective end devices is shown in the flow diagram 8.1.

In this master thesis embedded devices for AI applications from different manufacturers are used. Each manufacturer offers its own framework so that the hardware can run the CNN or provide the optimal performance. The Keras models are first converted into a Tensorflow format. Since, except for the GPU of the Jetson Nano, no hardware supports the Keras format. For the framework of Nvidia and Intel the Tensorflow saved model format is used, while for the EdgeTPU a Tensorflow Lite format is required. When converting to the saved model the structure of the model is still preserved, but when converting to the TFlite format the model gets optimized with a loss of information. A backwards compilation is therefore no longer possible. For the Openvino Framework the TF2 saved model is converted to a TF1.15 froozen model format. Since the NCS2 does not yet fully support the Intermediate Representation (IR) v11 and compiling to IRv6 causes problems with the saved model, which will not be explained further. Antonini et al. 2019; *TensorFlow Lite Converter* 2020; *Using the SavedModel Format | TensorFlow Core* 2020

To transfer the Tensorflow format to the respective AI engines the Framework APIs offer their own compilers. These are discussed in more detail in the respective subchapters.

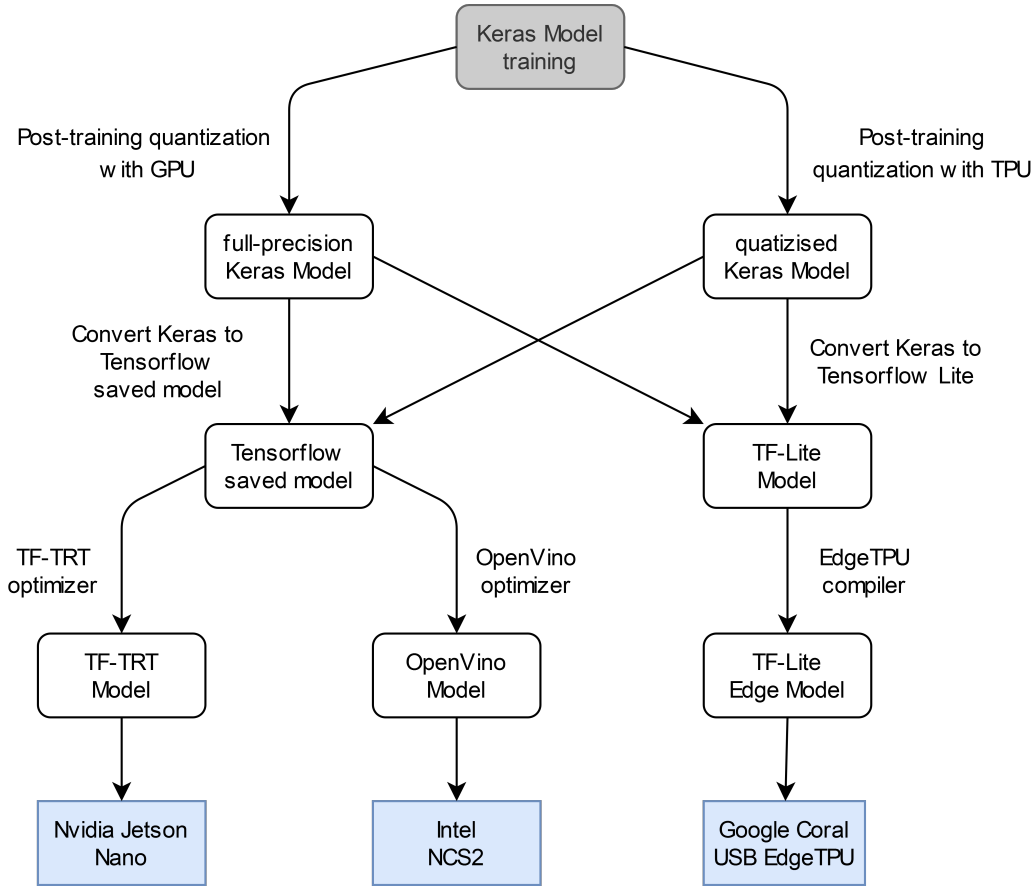


Figure 8.1: Work flow from Mobilenet-v2 training to different hardware specific models

8.1 Keras to TFlite for edge TPU

This subchapter deals with the compilation from TFlite to TFlite-edgeTPU, which is necessary for using Coral.

As described in the previous chapters, the edge TPU only supports uint8 arithmetic. The model parameters must therefore be compilable to full 8-bit interger arithmetic. Further conditions are constant tensor size and model parameters, as well as highest 3 dimensional tensors. For tensors with more than 3 dimensions, the inner ones must not have values greater than 1. Currently, the edge TPU does not support all tensorflow operations. The supported operations are published on the official Coral website *TensorFlow Models on the Edge TPU* 2020. When training and compiling the models the TF2 is used which sets the input and output of the model to float by default. Since the edge TPU can only handle 8-bit integer, the compiler outsources the input and output tensor to the host CPU. The compilation workflow is explained in the diagram 8.2.

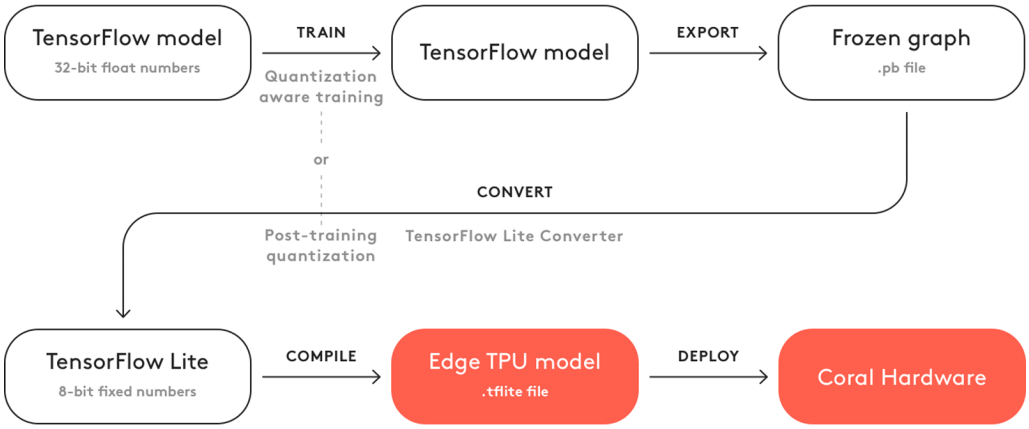


Figure 8.2: Workflow to compile Tensorflow model to TFlite-edgeTPU format for Google Coral hardware
Source: *TensorFlow Models on the Edge TPU* 2020

The following listings 8.1, 8.2, 8.3 contain the information that the compiler outputs after successful compilation. This information includes the time the compiler took to compile, the size of the input and output model and how much internal memory on the edge TPU and external memory on the Raspberry Pi is used by the model. Further information are the number of subgraphs and the total number of operations. The operations are organized into those performed on the edge TPU and those performed on the host CPU. Due to a problem in the compiler the EfficientNet-B0 (see listing: 8.1) could not be compiled at this time. According to Coral support, the bug will be fixed in the next update. Coral-Support 2020a

When comparing the compiling information from listing 8.2 and listing 8.3, the NASNet-Mobile with 759 operations and 7.64 MiB is significantly heavier than the MobileNet-v2 with 73 operations and 3.11 MiB. Accordingly, the compiler needs 7213ms to optimize the NASNet-Mobile model, while only 410ms are required for the MobileNet-v2. The Coral has only a small internal memory with 8 MiB. The MobileNet-v2 requires 3.33MiB for caching, so it fits completely on the memory. The NASNet-Mobile with its 6.31 MiB is too large for the internal memory, which is why 221.44 KiB are buffered externally. External buffering significantly reduces the performance of the model, since the latency increases when loading the model. Except for the input and output operations, all operations for the inference are executed on the TPU. The input and output operations are floating-point operations that are not supported by the TPU, so the calculation is done on the CPU.

EfficientNet-B0 compiling information

```
1 Edge TPU Compiler version 2.1.302470888
2
3 Internal compiler error. Aborting!
```

Listing 8.1: Compiling information from Efficientnet-B0 Keras model to TFlite Edge TPU Model

MobileNet-v2 compiling information

```
1      Edge TPU Compiler version 2.1.302470888
2
3      Model compiled successfully in 410 ms.
4
5      Input size: 3.06MiB
6      Output size: 3.11MiB
7      On-chip memory used for caching model parameters: 3.33MiB
8      On-chip memory remaining for caching model parameters: 4.39MiB
9      Off-chip memory used for streaming uncached model parameters: 0.00B
10     Number of Edge TPU subgraphs: 1
11     Total number of operations: 73
12
13     Model successfully compiled but not all operations are supported by the Edge TPU.
14     A percentage of the model will instead run on the CPU, which is slower.
15     If possible, consider updating your model to use only operations supported by the
           Edge TPU.
16
17     Number of operations that will run on Edge TPU: 71
18     Number of operations that will run on CPU: 2
19     See the operation log file for individual operation details.
```

Listing 8.2: Compiling information from Mobilenet-v2 Keras model to TFlite Edge TPU Model

NASNet-Mobile compiling information

```
Edge TPU Compiler version 2.1.302470888

Model compiled successfully in 7213 ms.

Input size: 5.47MiB
Output size: 7.64MiB
On-chip memory used for caching model parameters: 6.31MiB
On-chip memory remaining for caching model parameters: 0.00B
Off-chip memory used for streaming uncached model parameters: 221.44KiB
Number of Edge TPU subgraphs: 1
Total number of operations: 759


Model successfully compiled but not all operations are supported by the Edge TPU.
A percentage of the model will instead run on the CPU, which is slower.
If possible, consider updating your model to use only operations supported by the
    Edge TPU.

Number of operations that will run on Edge TPU: 757
Number of operations that will run on CPU: 2
See the operation log file for individual operation details.
```

Listing 8.3: Compiling information from NASNetMobile Keras model to TFlite Edge TPU Model

8.2 Keras To TF-TRT for Jetson Nano

In this section the optimization from Tensorflow saved model to Tensorflow TensorRT for the Jetson Nano is covered.

The Tegra GPU used in the Jetson Nano can be performed by the CUDA library from Nvidia in a general parallel task. Thus, most frameworks can be executed directly without compilation. To get the best possible performance, Nvidas TensorRT software development kit is used. TensortRT uses its own inference engine to execute the neural networks. Tensorflow models can not be put directly into a TensorRT format but TensorRT offers a Tensorflow-TRT container that allows the engine to run the optimized Tensorflow model from the box. To optimize the neural networks, layers are combined with each other and the distribution of the calculation to the CUDA cores is improved. This improves the energy and memory consumption as well as the latency. TensorRT allows parameter optimization to half precision 16-bit floating point or 8-bit integer like Coral. In this case the models are converted to FP16. *Nvidia TensorRT Developer-Guide* 2020; *Nvidia TF-TRT User-Guide* 2020

Certain parameters are set for the compilation, these can be seen in the listing 8.4. These are the used precision with FP16 and the maximum workspace with 100 MiB. The max_workspace parameter determines how much memory may be allocated for the execution of the CNN. The Tegra GPU of the Jetson Nano shares a 4Gb memory with the CPU, which is not sufficient for too much workspace. This has the consequence that the TensorRT engine cannot be generated and the model is executed with low performance as Tensorflow model. The Listing 8.5 shows the output information of the compiler, which is limited to the used TensorRT version. It is therefore not evident to what extent the models differ after compilation.

```
conversion_params = trt.DEFAULT_TRT_CONVERSION_PARAMS._replace( precision_mode=trt.
    TrtPrecisionMode.FP16,
max_workspace_size_bytes=100*1028)
```

Listing 8.4: Parameter settings for converting Keras saved models to TF-TRT model

```
Converting to TF-TRT FP16...
INFO:tensorflow:Linked TensorRT version: (5, 1, 5)
INFO:tensorflow:Loaded TensorRT version: (5, 1, 5)
INFO:tensorflow:Running against TensorRT version 5.1.5
```

Listing 8.5: Compiler information about used TensorRT version

8.3 Keras Openvino optimization

In this subchapter we discuss the optimization of the Tensorflow model for the Intel Openvino Framework.

The Openvino framework uses an Intermediate Representation of a network to read, load and execute models on the Inference engine. The openvino toolkit offers a model optimizer that transfers models into an IR across the framework. For an intermediate representation 2 files are used to describe the network. An .xml file to describe the network topology and .bin file which contains the weights and label data. For more information about the Intermediate Representation, see Duboscq et al. 2013,Cyphers et al. 2018 .The Openvino framework uses its own graph representation format and operation set. For the compilation the Openvino version 2020.1 is used which already has IR11. At compile time the NCS2 supports IR7 and the corresponding operation set, so the models are compiled on IR7 instead of IR11. The OpenvinoToolKit 2020.1 should be able to directly translate Tensorflow saved models to IR. In this project, however, the translation failed, it is assumed that it is a version conflict between TF2 and IR7, but the reason is not discussed further. As work around the TF2 saved model is compiled into a TF1.15 froozen model format, which can be translated to IR7 without problems. *Model Optimizer Developer Guide - OpenVINO™ Toolkit* 2020

When converting from TF2 saved model to TF1.15 froozen model we get the following information, the number of parameters which could be frozen and the number of parameters which were converted. For the MobileNet-v2 (see listing 8.6) 264 variables get converted, for the NASNet-Mobile (see listing 8.8) 1128 variables and for the EfficientNet-B0 (see listing 8.10) 313 variables.

The listings 8.7, 8.9 8.11 show the compile information of the Openvino optimizer. These consist of the IR version, the file name, compile time and the size or memory consumption of the source model. All models are translated to IR7 as described above. Just like the EdgeTPU, the NASNet-Mobile model takes significantly more time to compile and memory storage than the other models. Which is to be expected with considerably more parameters. Notice that the Openvino optimized models are significantly larger than the other edge devices.

MobileNet-v2 compiling information

```
1    INFO:tensorflow:Froze 264 variables.
2    INFO:tensorflow:Converted 264 variables to const ops.
```

Listing 8.6: Optimizer information from Mobilenet-v2 Keras model to Intel Openvino framework

```
1    Model Optimizer version:    2020.1.0-61-gd349c3ba4a
2
3    [ SUCCESS ] Generated IR version 7 model.
4    [ SUCCESS ] XML file: /frozen_model.xml
5    [ SUCCESS ] BIN file: /frozen_model.bin
6    [ SUCCESS ] Total execution time: 12.21 seconds.
7    [ SUCCESS ] Memory consumed: 317 MB.
```

Listing 8.7: Optimizer information from Mobilenet-v2 Keras model to Intel Openvino framework

NASNet-Mobile compiling information

```
1    INFO:tensorflow:Froze 1128 variables.
2    INFO:tensorflow:Converted 1128 variables to const ops.
```

Listing 8.8: Optimizer information from NASNetMobile Keras model to Intel Openvino framework

```
1    Model Optimizer version:    2020.1.0-61-gd349c3ba4a
2
3    [ SUCCESS ] Generated IR version 7 model.
4    [ SUCCESS ] XML file: /frozen_model.xml
5    [ SUCCESS ] BIN file: /frozen_model.bin
6    [ SUCCESS ] Total execution time: 52.51 seconds.
7    [ SUCCESS ] Memory consumed: 425 MB.
```

Listing 8.9: Optimizer information from NASNetMobile Keras model to Intel Openvino framework

EfficientNet-B0 compiling information

```
1    INFO:tensorflow:Froze 313 variables.
2    INFO:tensorflow:Converted 313 variables to const ops.
```

Listing 8.10: Optimizer information from Efficientnet-B0 Keras model to Intel Openvino framework

```
1    Model Optimizer version:    2020.1.0-61-gd349c3ba4a
2
3    [ SUCCESS ] Generated IR version 7 model.
4    [ SUCCESS ] XML file: /frozen_model.xml
5    [ SUCCESS ] BIN file: /frozen_model.bin
6    [ SUCCESS ] Total execution time: 19.72 seconds.
7    [ SUCCESS ] Memory consumed: 376 MB.
```

Listing 8.11: Optimizer information from Efficientnet-B0 Keras model to Intel Openvino framework

9 Experiment Information

In the following chapter the structure of the comparison experiment of AI acceleration devices is described.

The goal of this experiment is to test the performance of the peripheral devices Google Coral USB Accelerator, Nvidia Jetson Nano and the Intel Neuronal Compute Stick 2 with the 3 Convolutional Neuronal Network's for mobile applications MobileNet-v2, NASNet-Mobile and EfficientNet-B0. The following key figures are determined: The utilization of the CPU and RAM memory during the inference, the inference time per image, the power consumption and the resulting efficiency. Furthermore, a Confusion matrix is created from which the accuracy, F1-score, precision and recall are determined. The standard power supplies of the experimental units could influence the result by low power output. Therefore, a Damper S-50-5 power supply with 50W (5V - 10A) output line is used for a stable supply. For measuring the power consumption a TC66 USB-C power meter, illustrated in the picture 9.1, is used. The measuring device delivers voltage and ampere in a measuring cycle of 1s, which are recorded live from PC via Micro-USB. For the power consumption it is important to note that the test is performed directly at the device and not via a remote connection. The power consumption in idle is to be understood with the configuration of screen via HDMI and the input devices keyboard and mouse. Also note that the Python API of the AI accelerator is used for the experiment. The use with the C++ API is not considered.

Figure 9.1: TC66 USB C power meter to measure power consumption at the edge devices

The determination of the key data and the creation of the confusion matrix is done with a Python3.7 script. The same test data set with 2667 images that was used for testing the Keras models in Colab is used, which allows a better comparison of the results. During the experiment the timestamp before and after the inference is stored for each image, so the inference time per sample is calculated. For each inference the current CPU and RAM usage is collected in a list. To create the Confusion Matrix the result of the Inference determines the row and the Current Class determines the column. Whereby with a comparison of the result correct and wrong results are counted. At the end of each run the results are stored in an .CSV file. The Pythoncode of the Python script can be seen in the listing 9.1. To reduce the influence of fluctuations, 10 runs are performed per CNN model and end device. The recording of the power consumption is done externally with a USB-C meter.

```
# Loop throuht all classes from defined labels
for actual_class in labels:
    # Load and preprocess image
    image = readImage()
    input_data = preprocessImage(Image)
    # Get start time to measure the inference time
    start = time()
    # Get the One-Hot encoded result from the inference
    i_predict = classify_image(input_data)
    # Get stop time
    stop = time()
    # Get index from the actual_class
    j_actual = labels.index(actual_class)
    # Increase variable right if indices are same, else increase wrong
    if i_predict == j_actual:
        right += 1
    else:
        wrong += 1
    # Increase array field from both indicies
    confMatrix[i_predict][j_actual] += 1
    # Add time, CPU in % and Memory in % to lists
    time_list.append(stop-start)
    cpu_list.append(cpu)
    mem_list.append(mem)
```

Listing 9.1: Experiment code of the Python3.7 script to determine the performance data of the inference

The edge-TPU and the edge-GPU have additional power settings. The Coral offers the possibility to switch between a standard and maximum operation frequency of the TPU. With the standard frequency the power is halved which reduces the power consumption to 1W at full load for the TPU. The Jetson Nano offers a similar possibility, here a 5W and 10W mode can be set, which limits the GPU power to 5W or 10W. Note that not only the GPU frequency is reduced, but also the CPU and hardware accelerators. The 5W mode is set when used with a 5V-2A power supply or with multiple peripherals. In the experiment only the 10W mode is used to get the best performance from the system. For Coral both frequencies are taken into account, because the host CPU is not affected and the TPU is not loaded by using a batch size of 1.

10 Experiment Result of Coral USB Accelerator

This chapter evaluates the results of the experiment on the CUA with the MobileNet-v2 and NASNet-Mobile, running on TPU at standard and at maximum frequency. The EfficientNet-B0 can not be compiled at the time of the experiment and is therefore not considered. The evaluation includes the CPU and RAM memory consumption, inference time, power consumption, efficiency as well as the accuracy, F1-score, precision and recall calculated by the confusions matrix. As Host-CPU a Raspberry Pi 4 with 4GB RAM is used which communicates with the USB accelerator via the USB3 interface.

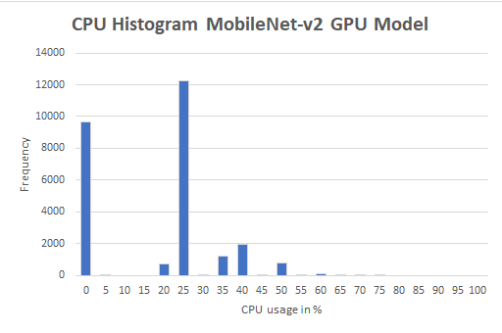
10.1 CPU Workload

This subchapter shows the utilization of the host CPU as a histogram in percent during the conclusion of the dataset on the Edge TPU. The CPU used is an ARM v8 quad-core Cortex-A72 64-bit SoC with 1.5GHz.

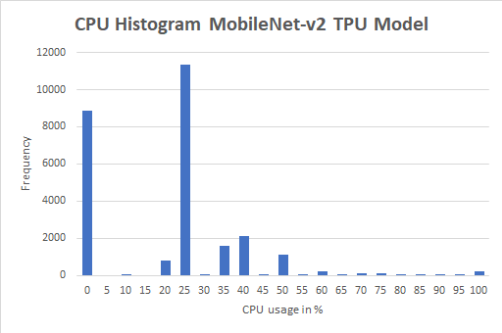
In the figures 10.1 and 10.3 the histograms of the CPU load on the MobileNet-v2 and the NASNet-Mobile with the standard and maximum frequency of the TPU can be seen. The bar charts 10.2 and 10.4 show the average CPU usage. The histograms of the MobileNet-v2 show an inconsistent CPU usage. The values jump between 0% and 25% at STD frequency and 0% and 25%-35% at MAX frequency. The NASNet-Mobile shows a more constant load. The values are in the interval of 15% and 45% utilization or 0%. Furthermore, the average utilization of the NASNet-Mobile with ∼16% is lower than the MobileNet-v2 with ∼21%. The operation frequency of the TPU hardly influences the workload of the CPU, although the input and output sensors are calculated on the CPU and not on the TPU. A difference between the GPU and TPU trained models is not visible in this test.

The results show the advantages of an external AI accelerator. The host CPU is rarely used in the conclusion. Since the calculations, except for the input and output of the CNN, take place completely on the edge-TPU. It is obvious that the CPU of the main board has no significant influence on the inference of the Coral USB Accelerator.

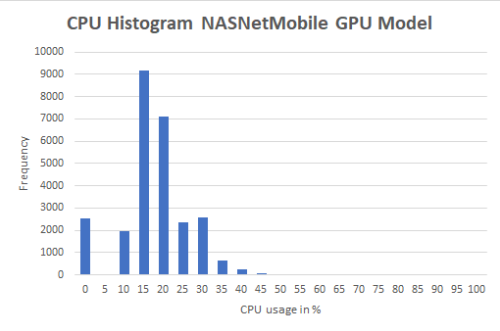
with standard frequency



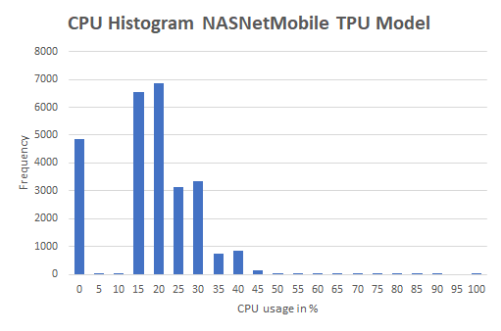
(a) Histogram of CPU workload on Coral USB Accelerator with MNV2 GPU model and standard operating frequency



(b) Histogram of CPU workload on Coral USB Accelerator with MNV2 TPU model and standard operating frequency



(c) Histogram of CPU workload on Coral USB Accelerator with NAS-M GPU model and standard operating frequency



(d) Histogram of CPU workload on Coral USB Accelerator with NAS-M TPU model and standard operating frequency

Figure 10.1: CPU workload histograms with standard operating frequency on Coral USB Accelerator

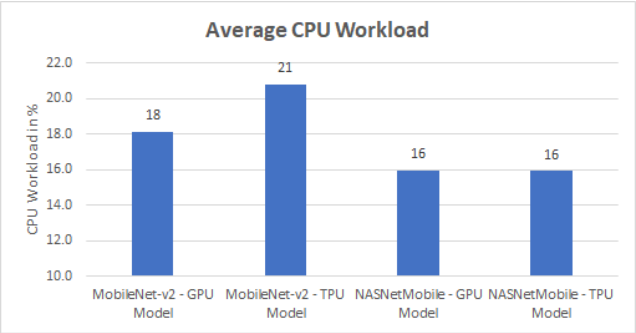
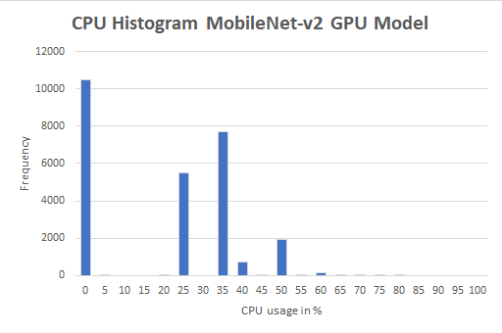
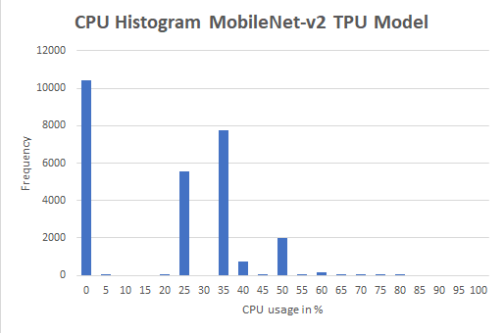


Figure 10.2: Average CPU Workload with different CNN-Models on Coral USB Accelerator and standard operating frequency

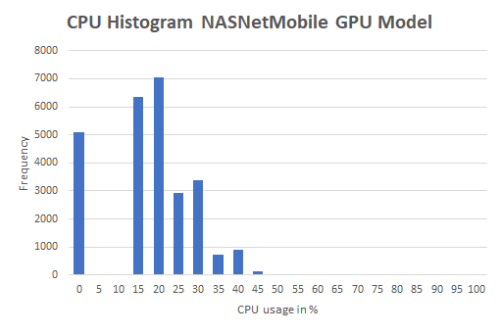
with maximal frequency



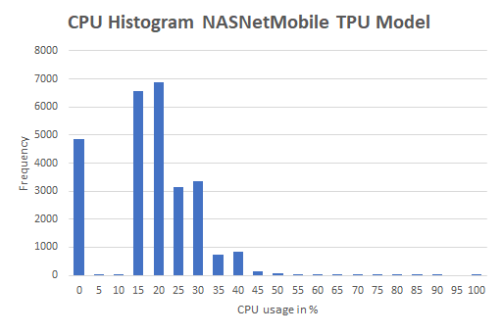
(a) Histogram of CPU workload on Coral USB Accelerator with MNV2 GPU model and maxium operating frequency



(b) Histogram of CPU workload on Coral USB Accelerator with MNV2 TPU model and maximum operating frequency



(c) Histogram of CPU workload on Coral USB Accelerator with NAS-M GPU model and maximum operating frequency



(d) Histogram of CPU workload on Coral USB Accelerator with NAS-M TPU model and maximum operating frequency

Figure 10.3: Histogram of CPU workload on Coral USB Accelerator with NAS-M TPU model and maximum operating frequency

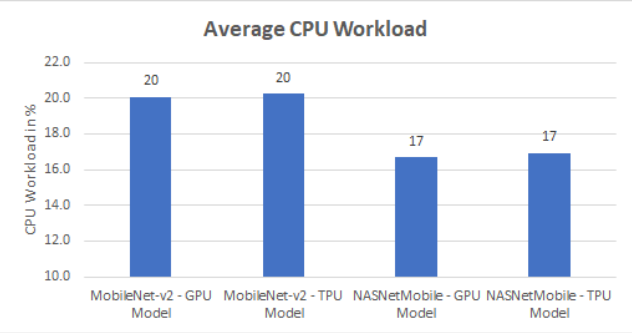


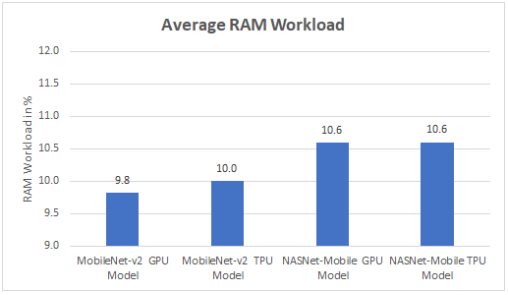
Figure 10.4: Average CPU Workload with different CNN-Models on Coral USB Accelerator and maxium operating frequency

10.2 Memory Workload

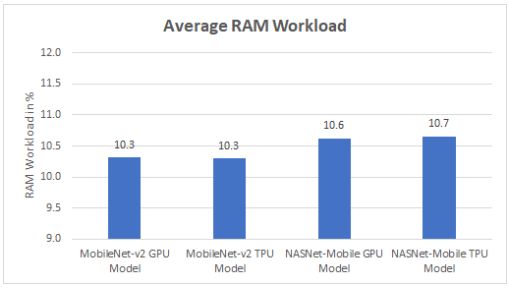
In this section the result of the RAM memory consumption is discussed. The CUA has its own small internal memory to store model data and reduce latency. If this is not enough, model data is stored in the external memory of the host, in this case the 4GB.

The diagrams 10.5a and 10.5b show the average memory consumption during the experiment in percent. The memory consumption differs between the models by only 0.7% which is 28MB. From the compilation of the models it is known that the MobileNet-v2 can be cached on the internal memory. While the NASNet-Mobile on the host memory occupies 221.44KiB (∼226KB), but in view of the 4GB memory the ∼226KB are not very relevant. Also the operation frequency does not show a big influence, only on the mobile net the consumption increases by 0.4% (16MB). As shown in the compilation, the GPU and TPU trained model do not differ in size, so as expected there is no difference in memory consumption.

The result of the memory consumption clearly shows the advantage of a USB AI accelerator. By shifting the computation to an external chip, the host system is less stressed and more resources are available for other tasks. It has to be taken into account that only 1 image per inference (batch size =1) is used in the experiment. The TPU is designed for larger amounts of data, so the memory consumption is very relevant for an optimal usage.



(a) with standard operating frequency



(b) with maximal operating frequency

Figure 10.5: Average memory Workload with different CNN-Models on Coral USB Accelerator

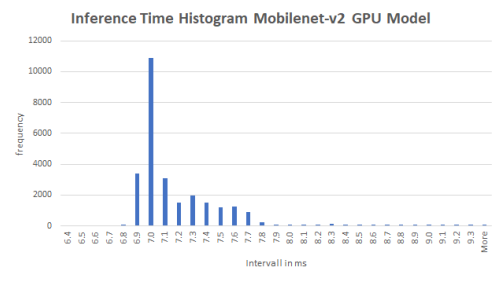
10.3 Inference Time

In the following subchapter the results of the Inference Time for the MobileNet-v2 and NASNet-Mobile are evaluated with the standard and maximum operations frequency.

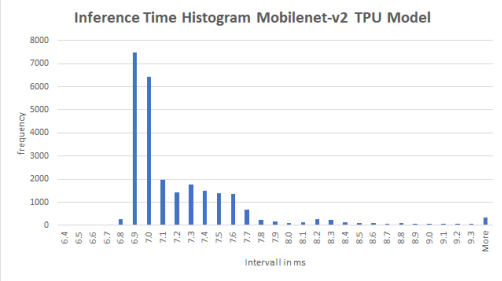
The determined time value for the inference is shown is ms with the STD frequency in the histograms 10.6 and with the MAX in the histograms 10.8. The average inference times of the models are shown in the figures 10.7 with STD frequency and 10.9 with MAX frequency. From the results it is clear that there is no significant difference between the different trained models. But you can clearly see the difference in performance between the MobileNet-v2 and the NASNet-Mobile. The MobileNet-v2 needs on average 7.1ms or 7.2ms in standard mode and 5.7ms in maximum mode. While the NASNet-Mobile needs 21ms in standard mode and 14.3ms on average, which is about 3 times more. The performance difference between the two operation frequencies standard and maximum is visible in both models. The average inference time is reduced by approx. ∼20% with the MobileNet-v2 and by approx. ∼30% with the NASNet-Mobile. This result agrees with the results of the MobileNet-v1 benchmark on CUA from the paper Libutti et al. 2020.

As expected, the result shows a lower inference time for the MobileNet-v2 with the lighter modelstructure of 73 operations and a total largest of 3.11MiB, than the more complex NASNet-Mobile with 759 operations and a size of 757MiB. Performance is also improved with the use of the maximum frequency of 20% for the MobileNet-v2 and 30% for the NASNet-Mobile. In both cases the Edge TPU is not fully utilized due to the single frame inference.

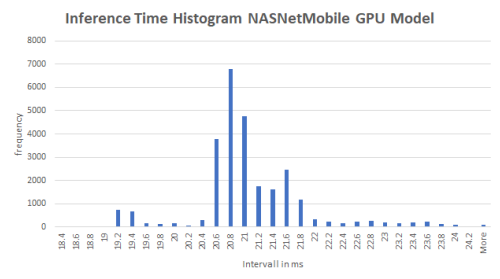
with standard frequency



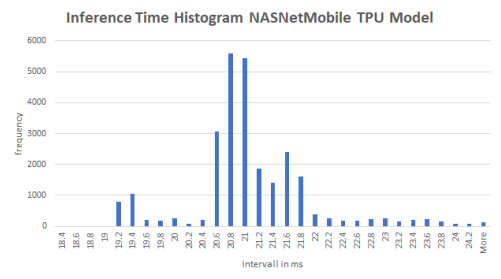
(a) Histogram of inference time on Coral USB Accelerator with MNV2 GPU model and standard operating frequency



(b) Histogram of inference time on Coral USB Accelerator with MNV2 TPU model and standard operating frequency



(c) Histogram of inference time on Coral USB Accelerator with NAS-M GPU model and standard operating frequency



(d) Histogram of inference time on Coral USB Accelerator with NAS-M TPU model and standard operating frequency

Figure 10.6: Inference time histograms with standard operating frequency on Coral USB Accelerator

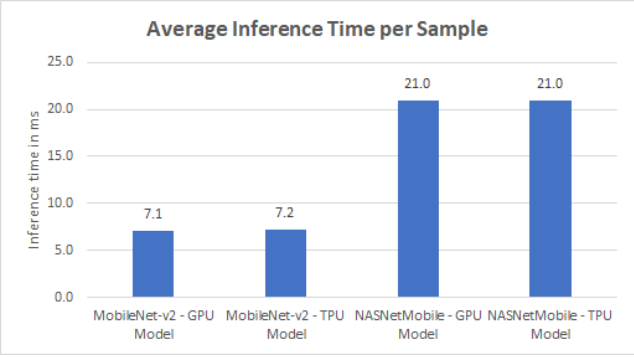
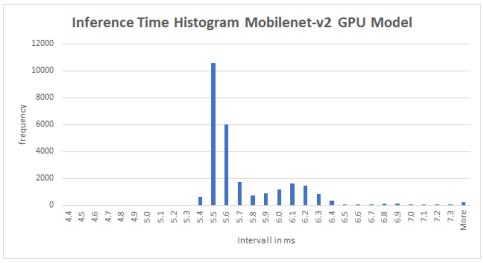
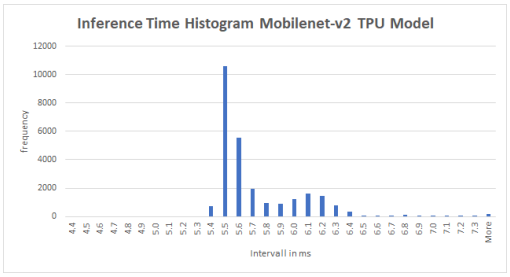


Figure 10.7: Average inference time with different CNN-Models on Coral USB Accelerator and standard operating frequency

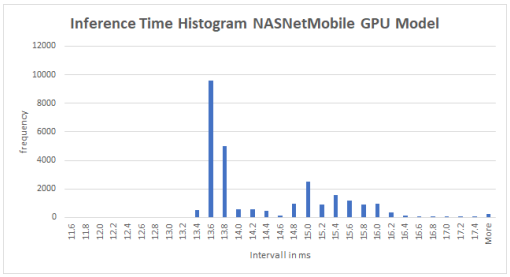
with maximal frequency



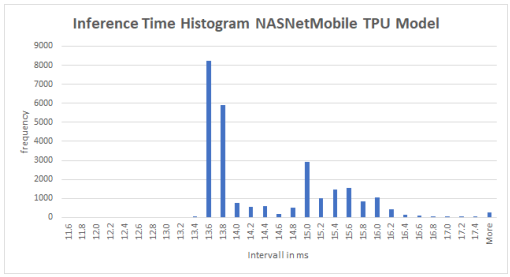
(a) Histogram of inference time on Coral USB Accelerator with MNV2 GPU model and maximum operating frequency



(b) Histogram of inference time on Coral USB Accelerator with MNV2 TPU model and maximum operating frequency



(c) Histogram of inference time on Coral USB Accelerator with NAS-M GPU model and maximum operating frequency



(d) Histogram of inference time on Coral USB Accelerator with NAS-M TPU model and maximum operating frequency

Figure 10.8: CPU workload histograms with standard operating frequency on Coral USB Accelerator

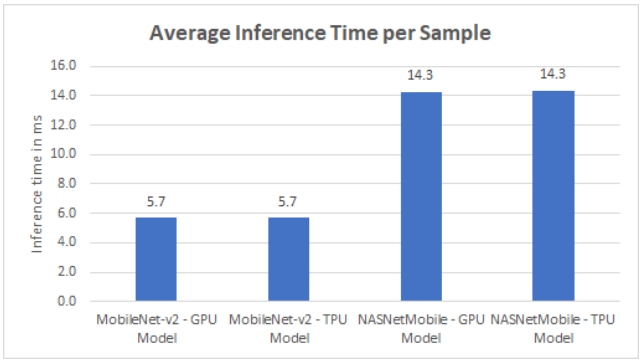


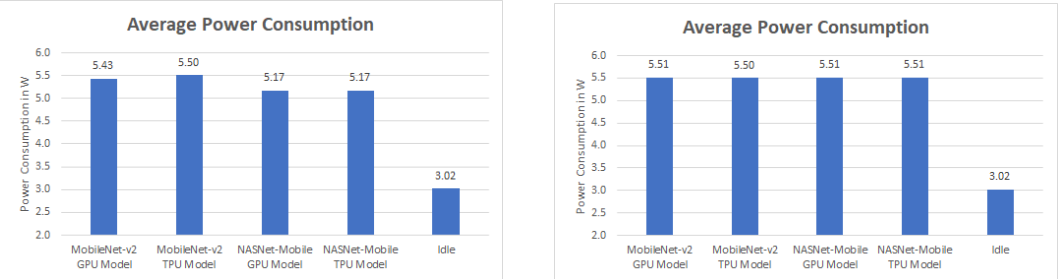
Figure 10.9: Average inference time with different CNN-Models on Coral USB Accelerator and maximum operating frequency

10.4 Power Consumption

The following section deals with the result of the power consumption of the MobileNet-v2 and the NASNet-Mobile with STD and MAX operation frequency. The measured values are obtained with a TC66 USB-C meter. The following equipment is used for the measurement, Raspberry Pi 4, Coral USB Accelerator, screen via HDMI mini and bluetooth receiver for mouse and keyboard.

The bar graphs 10.10a and 10.10b show the average power consumption of the MobileNet-v2 and the NASNet-Mobile for the two frequencies. Below the standard frequency it can be seen that the NASNet-Mobile with 5.17W consumes approximately ∼330mW less than the MobileNet-v2 with 5.42W or 5.5W. At the maximum frequency both models have a consumption of 5.5W. In idle mode the RPI4 needs 3.02W on average.

Due to the low batch size of 1, the Edge TPU is not continuously loaded and therefore the consumption hardly differs between the operation modes. This is also shown by the benchmark test of MobileNet-v1 Libutti et al. 2020. The difference in the NASNet-Mobile is probably due to the limited internal memory and the half operations frequency. When compiling the NASNet-Mobile you can see that the 221.44KiB of the model is reloaded. By reloading the model data, the TPUs are more often in a waiting state and therefore the average consumption is reduced. Officially this assumption is not confirmed, Coral Support points out the reduction of memory and frequency in STD mode. The architectures and the libary of the Edge-TPU are not open-source, therefore no further information are given. Coral-Support 2020b



(a) with standard operating frequency

(b) with maximal operating frequency

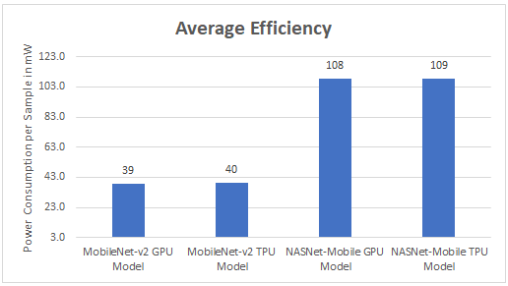
Figure 10.10: Average power consumption with different CNN-Models on Coral USB Accelerator

10.5 Efficiency

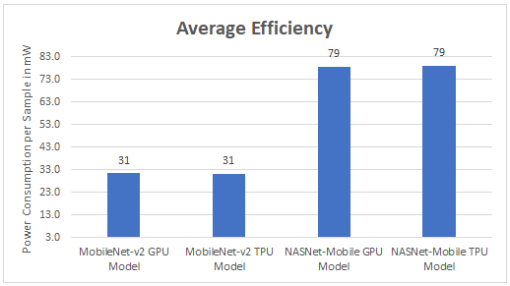
The following section discusses the efficiency of the models MobileNet-v2 and NASNet-Mobile with half operations frequency and full operations frequency. The specification is how much power in mW is needed to classify an image.

The two diagrams in 10.11a and 10.11b display the power consumption of a sample for the MobileNet-v2 and NASNet-Mobile. The diagram 10.11a represents the result for with STD frequency. The MobileNet-v2 consumes an average of 39-40 mW for an image and the NASNet-Mobile 108-109mW. The efficiency in MAX mode is shown in figure 10.11b. The consumption is reduced to 31mW for the MobileNet-v2 and to 79mW for the NASNet-Mobile.

The result of the experiment shows that in this case with the Coral USB Accelerator the best efficiency is achieved with the MobileNet-v2 and at the maximum operations frequency. It is obvious that the MobileNet-v2 has a much better efficiency than the NASNet-Mobile, because it needs approximately only one third of the energy for a classification. Furthermore it is recognizable that in MAX mode the consumption per sample is reduced by about 25%. The efficiency of Graphics Processing Unit trained and Tensor Processing Unit trained models are identical.



(a) with standard operating frequency



(b) with maximal operating frequency

Figure 10.11: Average efficiency with different CNN-Models on Coral USB Accelerator

10.6 Confusion Matrix

In the following section the confusion matrices of the neural network models MobileNet-v2 and NASNet-Mobile are shown in GPU and TPU trained version. To create the confusions matrix, a test data set with 2667 images of the CUA is classified. The result determines the row and the right class determines the column. Furthermore, the accuracy and the F1 score of the respective Convolutional Neuronal Networks are determined.

The figure 10.12 represents the 4 confusions matrices of the tested CNN models. The best accuracy value 0.935 and macro F1-Score 0.933 are calculated with the GPU trained model MobileNet-v2 10.12a. Least accurate is the TPU trained NASNet-Mobile model with an accuracy of 0.876 and a macro-F1 of 0.881. The TPU trained MobileNet-v2 with an accuracy of 0.907 and macro-F1 0.899 and the GPU trained NASNet-Mobile with an accuracy of 0.905 and macro-F1 0.914 show a similar performance. It is apparent that the NASNet-Mobile models the macro-F1 is higher than the accuracy. With the MobileNet-v2 models, on the other hand, the accuracy is higher than macro-F1.

In the comparison of the two model architectures the MobileNet-v2 performs better than the NASNet-Mobile. This is reflected in the fact that the MobileNet-v2 construction allows a better distinction between the classes glass bottle and plastic bottle than the NASNet-Mobile construction. This is shown by the macro-F1 score between the MobileNet-v2 and NASNet-Mobile architectures. For the NASNet-Mobile models, the accuracy is lower than the F1 score, while for the MobileNet-v2 models it is the other way round. This is an indication that one class influences the result, in this case the glass bottle. Furthermore, it can be seen that the GPU trained models have better accuracy values and macro F1-score than the TPU models.

MobileNet-v2 with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	62	2	1	1	1	4	0	2	0	1	0
food_bowl	1	160	0	0	0	0	0	2	0	0	0
food_box	5	3	218	1	0	2	0	1	0	1	1
glas_bottle	0	1	0	232	14	0	5	2	0	0	0
metal_can	0	0	0	6	228	0	0	0	0	0	0
plastic_bag	0	0	1	3	0	237	0	3	0	0	0
plastic_bottle	0	3	1	29	4	1	387	22	0	0	12
plastic_cup	1	10	0	1	0	0	3	328	1	0	0
plastic_cutlery	2	3	2	2	4	1	1	0	244	0	0
snack_wrap	0	0	1	0	0	0	0	0	0	219	0
tetrapack	0	0	1	2	0	0	0	0	0	3	178

right	2493
wrong	174
accuracy	0.935
macro-F1	0.933

(a) Confusion matrix on Coral USB Accelerator with MNV2 GPU trained model

MobileNet-v2 with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	5	5	7	2	7	0	7	4	2	1
food_bowl	1	166	1	0	0	0	0	8	2	0	0
food_box	2	0	209	0	0	0	0	0	1	1	0
glas_bottle	0	0	0	181	3	0	14	3	0	0	1
metal_can	0	2	1	5	240	0	3	1	1	1	3
plastic_bag	0	1	2	3	0	237	0	9	0	2	0
plastic_bottle	1	4	0	51	0	0	371	11	1	1	2
plastic_cup	0	3	0	4	0	0	0	321	0	0	0
plastic_cutlery	0	1	4	0	0	0	1	0	236	0	0
snack_wrap	0	0	3	5	1	1	1	0	0	210	2
tetrapack	0	0	0	21	5	0	6	0	0	7	182

right	2420
wrong	247
accuracy	0.907
macro-F1	0.899

(b) Confusion matrix on Coral USB Accelerator with MNV2 TPU trained model

NASNet-Mobile with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	0	0	1	0	3	0	1	1	0	1
food_bowl	0	162	0	1	0	0	0	3	1	0	0
food_box	1	2	218	0	0	3	0	0	0	3	0
glas_bottle	1	0	0	184	11	0	14	0	0	0	0
metal_can	0	0	0	13	233	0	1	0	0	0	0
plastic_bag	0	0	1	0	0	231	0	1	0	0	0
plastic_bottle	1	2	2	66	4	2	357	19	1	3	5
plastic_cup	1	14	2	12	0	6	21	336	1	2	0
plastic_cutlery	0	1	1	0	0	0	0	0	241	1	0
snack_wrap	0	1	1	0	0	0	0	0	0	203	3
tetrapack	0	0	0	0	3	0	3	0	0	12	182

right	2414
wrong	253
accuracy	0.905
macro-F1	0.914

(c) Confusion matrix on Coral USB Accelerator with NAS-M GPU trained model

NASNet-Mobile with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	68	3	12	1	0	1	2	3	0	1	1
food_bowl	0	162	1	0	0	0	0	1	0	0	0
food_box	0	0	190	0	0	0	0	0	0	0	0
glas_bottle	0	0	0	127	0	0	6	1	0	0	0
metal_can	0	1	1	14	236	0	0	0	0	4	2
plastic_bag	1	0	4	0	0	236	0	4	0	1	0
plastic_bottle	0	2	4	124	13	5	383	29	2	4	23
plastic_cup	0	13	7	3	1	3	5	322	1	2	0
plastic_cutlery	1	1	2	2	0	0	0	0	242	0	0
snack_wrap	1	0	4	0	0	0	0	0	0	207	2
tetrapack	0	0	0	6	1	0	0	0	0	5	163

right	2336
wrong	331
accuracy	0.876
macro-F1	0.881

(d) Confusion matrix on Coral USB Accelerator with NAS-M TPU trained model

Figure 10.12: Confusions matrix of the MobileNet-v2 and NASNet-Mobile trained with GPU and TPU accelerator and compiled for the Coral USB Accelerator

10.7 Precision, Recall and F1-score

In this subchapter the parameters precision, recall and F1-score determined from the confusions matrix are evaluated. The precision describes how often the prediction of the current class is true while the recall describes how often the current class is correctly classified as such. The F1-score is the harmonic average of precision and recall

The two matrices 10.13a and 10.13b show the determined precision and recall of the MobileNet-v2 and the NASNet-Mobile. It can be seen that the TPU trained models have a low precision, MobileNet-v2 0.626 and NASNet-Mobile 0.739, in the background class. Furthermore, it can be seen that the NASNet-Mobile networks classify plastic bottles with a low precision. At recall the class glass bottle is the most striking. The recall value is low with 0.458 for the TPU trained and 0.664 for the GPU trained NASNet-Mobile models as well as for the MobileNet-v2 models with 0.653 with TPU accelerator and 0.838 with GPU accelerator. In the F1 score matrix representation 10.14 the values of the harmonic mean of precision and recall are presented. The GPU trained MobileNet-v2 does not fall below the value 0.85 for any class, the TPU trained model shows the values of the class background with 0.753 and the class glass bottle with 0.756. For the GPU trained NASNet-Mobile model, the glass bottle class depicts a low score of 0.756 and for the TPU trained model the classes glass bottle with 0.618 and plastic bottles with 0.775 shows a bad F1 score.

The results demonstrate that the models created with the TPU accelerator often do not recognize objects correctly and classify them as background, whereas the background is usually recognized correctly. Furthermore, all models have difficulties to classify glass bottles correctly, mostly they are recognized as plastic bottles. Due to the similar features like shape and transparency this behaviour is expected.

Precision and Recall

Precision						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.838	0.626	0.905	0.739	None	None
food_bowl	0.982	0.933	0.970	0.988	None	None
food_box	0.940	0.981	0.960	1.000	None	None
glas_bottle	0.913	0.896	0.876	0.948	None	None
metal_can	0.974	0.934	0.943	0.915	None	None
plastic_bag	0.971	0.933	0.991	0.959	None	None
plastic_bottle	0.843	0.839	0.773	0.650	None	None
plastic_cup	0.953	0.979	0.851	0.902	None	None
plastic_cutlery	0.942	0.975	0.988	0.976	None	None
snack_wrap	0.995	0.942	0.976	0.967	None	None
tetrapack	0.967	0.824	0.910	0.931	None	None

(a) Precision matrix of the CUA compiled models trained with GPU and TPU accelerator

Recall						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.873	0.944	0.944	0.958	None	None
food_bowl	0.879	0.912	0.890	0.890	None	None
food_box	0.969	0.929	0.969	0.844	None	None
glas_bottle	0.838	0.653	0.664	0.458	None	None
metal_can	0.908	0.956	0.928	0.940	None	None
plastic_bag	0.967	0.967	0.943	0.963	None	None
plastic_bottle	0.977	0.937	0.902	0.967	None	None
plastic_cup	0.911	0.892	0.933	0.894	None	None
plastic_cutlery	0.996	0.963	0.984	0.988	None	None
snack_wrap	0.978	0.938	0.906	0.924	None	None
tetrapack	0.932	0.953	0.953	0.853	None	None

(b) Recall matrix of the CUA compiled models trained with GPU and TPU accelerator

Figure 10.13: Precision and recall matrix of the CUA compiled models trained with GPU and TPU accelerator

F1-score

F1-Score						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.855	0.753	0.924	0.834	None	None
food_bowl	0.928	0.922	0.928	0.936	None	None
food_box	0.954	0.954	0.965	0.916	None	None
glas_bottle	0.874	0.756	0.756	0.618	None	None
metal_can	0.940	0.945	0.936	0.927	None	None
plastic_bag	0.969	0.950	0.967	0.961	None	None
plastic_bottle	0.905	0.885	0.832	0.778	None	None
plastic_cup	0.932	0.933	0.890	0.898	None	None
plastic_cutlery	0.968	0.969	0.986	0.982	None	None
snack_wrap	0.986	0.940	0.940	0.945	None	None
tetrapack	0.949	0.883	0.931	0.891	None	None
macro-F1	0.933	0.899	0.914	0.881	None	None

Figure 10.14: F1-score matrix of the CUA compiled models trained with GPU and TPU accelerator

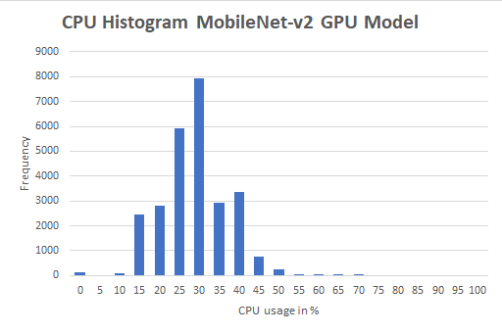
11 Experiment Result of Nvidia Jetson Nano

In the following chapter the results of the experiment on Nvidia Jetson Nano developer board with the CNN models MobileNet-v2, NASNet-Mobile and EfficientNet-B0 are evaluated in both GPU and TPU versions. Only the 10W mode of the Jetson Nano is considered to get the best performance. Note that the 4GB memory is shared by the CPU and the GPU. The results show the CPU and memory usage during inference, inference time, power consumption and the resulting efficiency. Furthermore, the resulting confusion matrices of the models are used to determine the accuracy, the macro F1 score, the precision, the recall and the F1 score for all classes.

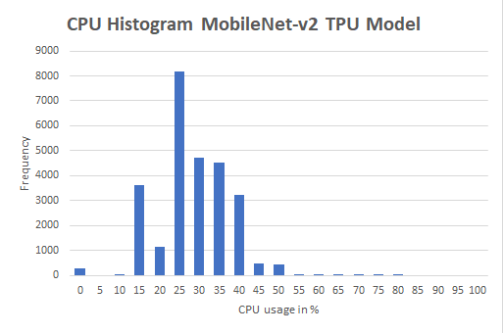
11.1 CPU Workload

This section deals with the determined CPU workload during the experiment. A quad-core ARM Cortex-A57 64-bit with 1.42 Ghz is used as CPU for the Jetson Nano. The measured values are displayed as a histogram and the average load as a bar chart.

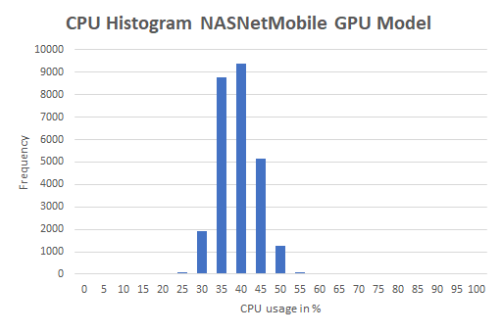
The measured values of the CPU utilization are visible as histogram in the figure 11.1. The average workload can be read in the illustration 11.2. According to the histograms 11.1a and 11.1b, the utilization of the CPU ranges from 15% to 50% for the MobileNet-v2 models. The histograms 11.1c and 11.1d indicate an interval from 20% to 50% for the NASNet-Mobile models. The EfficientNet-B0 is like the MNV2 in the range of 15% and 50%, see histogram 11.1e and 11.1f. The average values for the MNV2 are 27%, for the NAS-M around 37.5% and for the EffNet about 29.5%.



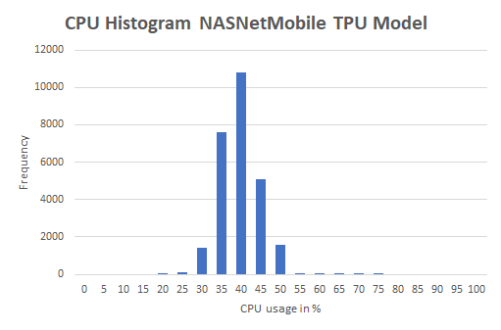
(a) Histogram of CPU workload on Jetson Nano with MNV2 GPU model



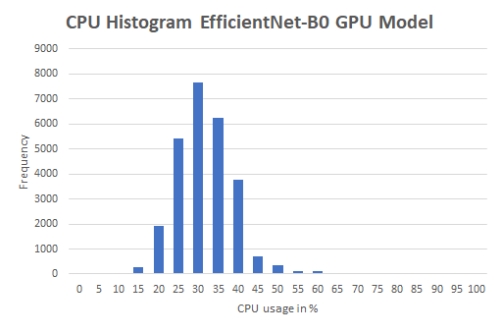
(b) Histogram of CPU workload on Jetson Nano with MNV2 TPU model



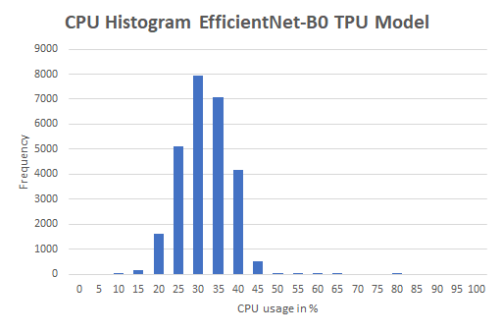
(c) Histogram of CPU workload on Jetson Nano with NAS-M GPU model



(d) Histogram of CPU workload on Jetson Nano with NAS-M TPU model



(e) Histogram of CPU workload on Jetson Nano with EffNet GPU model



(f) Histogram of CPU workload on Jetson Nano with EffNet TPU model

Figure 11.1: CPU workload histograms on Jetson Nano

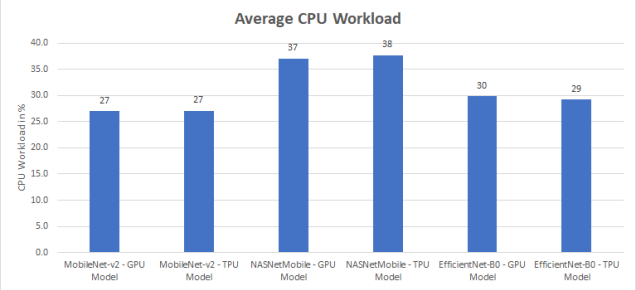


Figure 11.2: Average CPU Workload with different CNN-Models on Jetson Nano

11.2 Memory Workload

This section discusses the memory consumption in percent during the inference with the CNNs MobileNet-v2, NASNet-Mobile and EfficientNet-B0. The Jetson Nano uses a 4GB LPDDR4 1600MHz RAM used by the CPU and the GPU.

The diagram 11.3 indicates the consumption of the 4GB memory of the Jetson Nano. The load for the MNV2 is 85-86%, for the NAS-M 90-92% and for the EffNet 92-98%. The memory of the Jetson Nano is quite exhausted in the inference compared to the other end devices.

However, the memory of the GPU is additionally occupied by the CPU in the case of the Jetson Nano. Therefore it is not possible to determine exactly how much memory the respective models need. In general the memory consumption depends on the size of the CNN which is consistent with the result. The result demonstrates that the memory for the GPU is a main element of performance and is highly loaded.

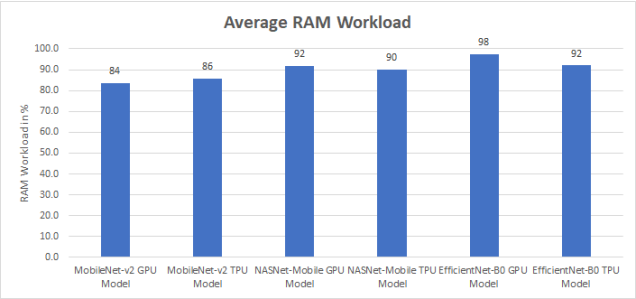


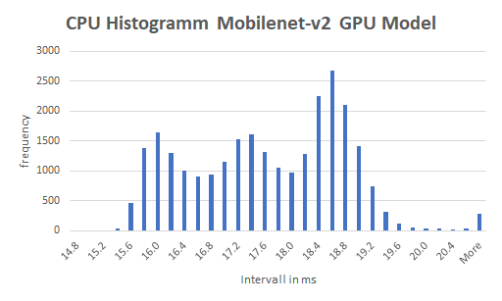
Figure 11.3: Average memory Workload with different CNN-Models on Jetson Nano

11.3 Inference Time

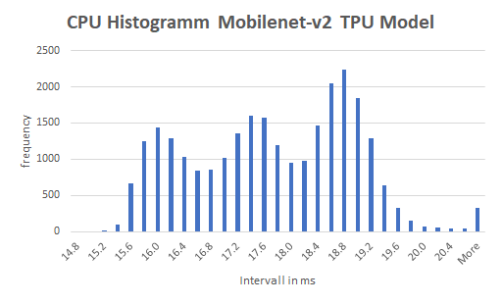
The following section evaluates the result of the measured Inference Time of the Convolutional Neuronal Network MobileNet-v2, NASNet-Mobile and EfficientNet-B0 on the Jetson nano. The results are listed in ms and can be seen in the subsequent histogram. The experiment has been run in 10W full power mode.

The figure 11.4 contains the histograms of the inference time of the edge-CNNs models in the GPU and TPU trained version. The classification times varies for all models within a range of 5ms. The MobileNet-v2 has the best result, with inference times ranging primarily between 15ms and 20ms, see 11.4a and 11.4b. This is the situation for the GPU and TPU models. The inference times of the NASNet-Mobile model are in most cases in the range of 44ms and 49ms and are the highest times of the models, 11.4c and 11.4d. The EfficientNet-B0 is positioned between the MNV2 and NAS-M with a spread of 32ms to 38ms, 11.4e and 11.4f. In the representation 11.5 the average inference time of the CNNs is visible. The MNV2 with an average time of 17.6ms takes only half as much time to predict an image as the EffNet with 35ms and a third of the time as the NAS-M with 48ms.

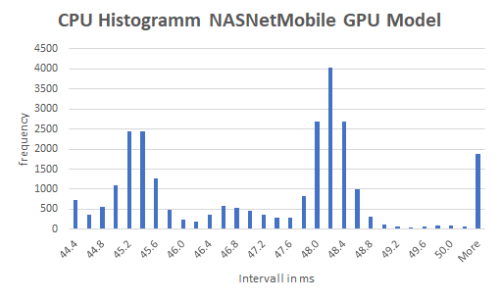
The results demonstrate a very variable behavior of the embedded GPU at the operations times. There is no difference between the GPU and TPU trained versions of the models. As expected, the NAS-M that has the most parameters to compute, needs most for a classification. While the number of parameters is only one criterion for the duration of the inference, it is recognizable that the structure of the CNN has a significant influence on the performance. Thus, the EfficientNet-B0 has similarly numerous parameters to be calculated as the NAS-M, but needs much less time. The histograms also show that especially with the larger models the inference time can take extreme values. It is assumed that these extreme values are caused by the small shared 4GB memory.



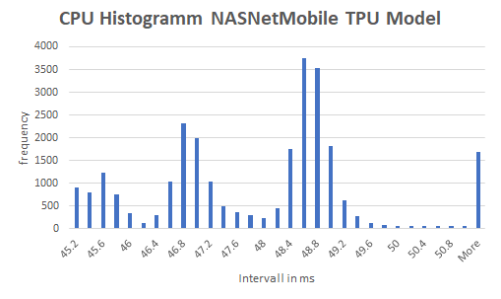
(a) Histogram of inference time on Jetson Nano with MNV2 GPU model



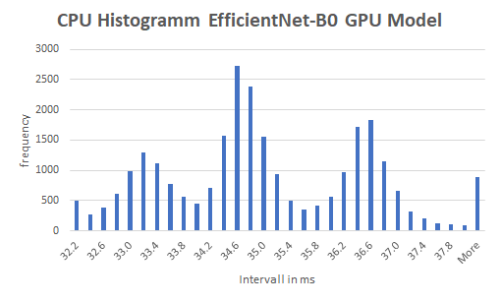
(b) Histogram of inference time on Jetson Nano with MNV2 TPU model



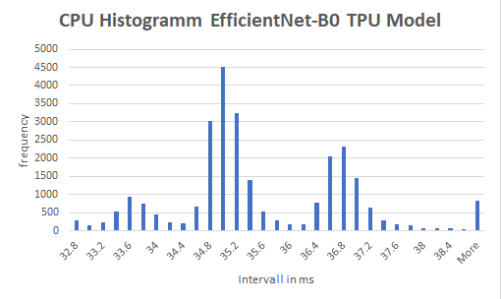
(c) Histogram of inference time on Jetson Nano with NAS-M GPU model



(d) Histogram of inference time on Jetson Nano with NAS-M TPU model



(e) Histogram of inference time on Jetson Nano with EffNet GPU model



(f) Histogram of inference time on Jetson Nano with EffNet TPU model

Figure 11.4: Inference time histograms of the MNV2, NAS-M and EffNet running on the Jetson Nano

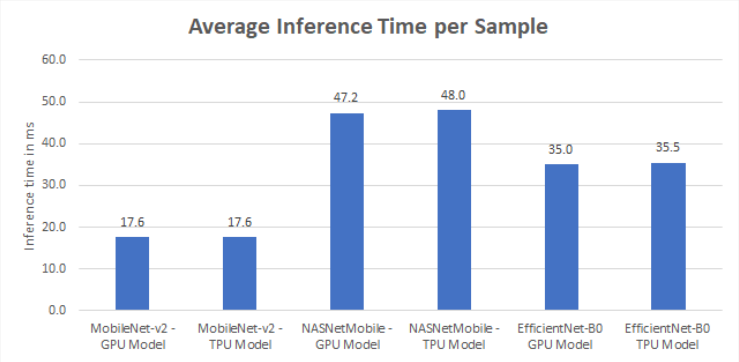


Figure 11.5: Average inference time with different CNN-Models on Jetson Nano

11.4 Power Consumption

This section explains the energy consumption of the Jetson Nano during the experiment with the MobileNet-v2, NASNet-Mobile and EfficientNet-B0 and respectively as GPU and TPU trained models. The edge-GPU is operated in 10W power mode.

The bar chart 11.6 illustrates the average consumption of the Convolutional Neuronal Network's during the inference on the Jetson Nano. The MNV2 needs as GPU version 7.52W and as TPU version 7.68W on average. For the NAS-M 8.76W for the GPU model and 8.98W for the TPU model are calculated as mean values. For the EffNet model the consumption is 8.51W for the GPU model and 8.42W for the TPU model as an average. In idle mode 3.37W is required.

As a result of the experiment it is obvious that the model structure and the size of CNNs have an influence on the power consumption of the edge-GPU. The MNV2 with the least parameters consumes the least, while the NAS-M with the most parameters consumes the most, followed by the EffNet. There is no significant difference between the models trained with GPU and TPU.

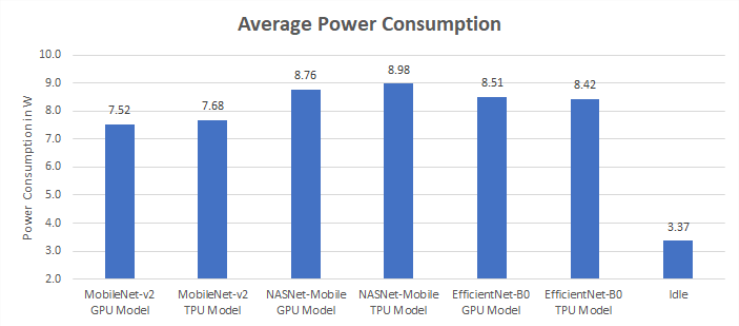


Figure 11.6: Average power consumption with different CNN-Models on Jetson Nano

11.5 Efficiency

In this subchapter the efficiency of the MNV2, NAS-M and EffNet models represented. The efficiency is calculated from the inference time and the energy consumption. Therefore, the result is the power consumption per sample and is given in mW.

The efficiency describes how much power is needed to classify a single image. The calculated result is illustrated in the figure 11.7 in mW by sample. With 132mW for the GPU and 135mW for the TPU model, the MobileNet-v2 needs much less power to predict an image. The NASNet-Mobile has the highest consumption per image with 414mW (GPU) and 43mW (TPU). With 298mW (GPU) and 299mW (TPU) per sample, the EfficientNet-B0 ranks between the two models.

The advantage of the MobileNet-v2 can be clearly seen from the result of the efficiency which is the ratio of energy consumption and inference time. The MNV2 requires on average only one third of the energy per prediction than the NAS-M and about 55% less than the EfficientNet-B0. The GPU and TPU trained models have similar performance.

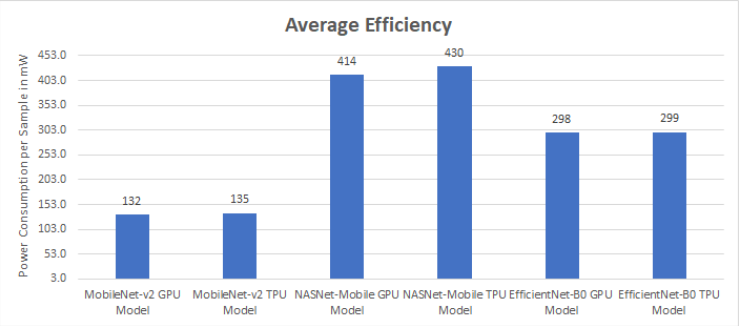


Figure 11.7: Average efficiency with different CNN-Models on Jetson Nano

11.6 Confusion Matrix

In the following subchapter the output of the determined confusion matrices for the models MobileNet-v2, NASNet-Mobile and EfficientNet-B0, each as a model trained with a cloud GPU and TPU is presented. The matrix contain 2667 images with 11 classes. From the Confustion Matrices the following parameters are calculated: the accuracy, the macro-F1, precision and recall as well as the F1 score for each class.

The figure 11.8 shows the 6 confusion matrices of the used models MNV2, NAS-M and EffNet in GPU and TPU version, respectively. The columns represent the current class and the row the predicted class. Furthermore, the accuracy and the macro-F1 are determined, both key figures describe the quality of the CNN. Whereby the macro-F1 quantifies the sum of the F1-score of the individual classes and describes the quality better in case of uneven matrices. The EfficientNet-B0 trained with the GPU 11.8e gives the best values with a accuracy of 0.952 and a macro-F1 of 0.951, followed by the TPU trained model 11.8f with a accuracy of 0.942 and a macro-F1 of 0.939. Good results are also given by the MobileNet-v2 with a accuracy of 0.936 on the GPU trained model 11.8a and 0.912 on the TPU trained model 11.8b. The macro-F1 is determined with 0.934 for the GPU trained model and 0.904 for the TPU trained model. The NAS-M provides the lowest result with an accuracy of 0.915 (GPU trained 11.8c) or 0.889 (TPU trained 11.8d) and a macro-F1 of 0.921 (GPU trained) or 0.893 (TPU trained).

From the confusion matrices it can be seen that the GPU versions of the models perform better in the classification of the images than the TPU trained models. Especially the glass bottle class turns out to be difficult to classify, due to the similarity with the plastic bottles. The NAS-M TPU trained model 11.8d performs worst. It can be observed that the accuracy of the MNV2 and EffNet models is minimally higher than the macro-F1, while the NAS-M models have a slightly higher accuracy than the macro-F1. That the values are close together is due to the similar number of images per class.

MobileNet-v2 with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	63	1	2	0	0	4	0	1	0	3	0
food_bowl	1	156	0	0	0	0	0	1	0	0	0
food_box	4	2	215	2	0	1	0	1	0	1	0
glas_bottle	0	0	0	214	11	0	4	1	0	0	0
metal_can	0	1	0	10	239	0	0	0	0	0	0
plastic_bag	0	0	1	3	0	238	0	1	0	0	0
plastic_bottle	0	3	2	38	1	1	387	15	0	0	8
plastic_cup	1	15	0	2	0	0	4	340	1	0	0
plastic_cutlery	2	4	3	3	0	1	1	0	244	0	0
snack_wrap	0	0	2	0	0	0	0	0	0	218	0
tetrapack	0	0	0	5	0	0	0	0	0	2	183

right	2497
wrong	170
accuracy	0.936
macro-F1	0.934

(a) Confusion matrix on Jetson Nano with MNV2 GPU model

MobileNet-v2 with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	66	5	3	4	2	4	1	6	2	2	2
food_bowl	1	167	1	0	0	0	0	9	2	0	0
food_box	2	0	212	0	0	0	0	0	1	1	0
glas_bottle	0	0	0	177	2	0	14	2	0	0	1
metal_can	0	2	0	10	242	0	1	1	0	0	3
plastic_bag	1	1	3	4	0	239	0	8	0	1	0
plastic_bottle	1	3	0	48	0	1	375	9	1	1	2
plastic_cup	0	3	0	4	0	0	0	325	0	0	0
plastic_cutlery	0	1	2	1	0	0	1	0	239	0	0
snack_wrap	0	0	4	5	1	1	0	0	0	209	1
tetrapack	0	0	0	24	4	0	4	0	0	10	182

right	2433
wrong	234
accuracy	0.912
macro-F1	0.904

(b) Confusion matrix on Jetson Nano with MNV2 TPU model

NASNet-Mobile with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	63	0	0	1	0	2	0	1	1	1	1
food_bowl	2	170	1	1	0	0	0	2	1	0	0
food_box	1	1	218	0	0	1	0	0	0	2	0
glas_bottle	2	0	0	197	10	0	19	0	0	0	0
metal_can	0	0	0	12	239	0	0	0	0	0	0
plastic_bag	0	0	1	0	0	237	0	1	0	0	0
plastic_bottle	1	1	2	60	1	2	354	22	0	1	5
plastic_cup	2	9	1	6	0	2	19	334	1	1	0
plastic_cutlery	0	0	1	0	0	0	1	0	242	1	0
snack_wrap	0	1	1	0	0	1	0	0	0	204	2
tetrapack	0	0	0	0	1	0	3	0	0	14	183

right	2441
wrong	226
accuracy	0.915
macro-F1	0.921

(c) Confusion matrix on Jetson Nano with NAS-M GPU model

NASNet-Mobile with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	1	9	1	0	1	1	2	0	1	1
food_bowl	0	169	0	0	0	0	0	2	0	0	0
food_box	0	0	194	0	0	0	0	0	0	0	0
glas_bottle	0	0	0	131	1	0	11	1	0	0	3
metal_can	0	1	0	12	235	0	0	0	0	2	1
plastic_bag	1	0	4	0	0	238	0	3	0	1	0
plastic_bottle	0	1	3	114	11	3	381	19	0	1	17
plastic_cup	0	8	6	3	1	3	3	332	1	1	1
plastic_cutlery	1	2	3	7	0	0	0	1	244	0	0
snack_wrap	2	0	5	0	0	0	0	0	0	214	1
tetrapack	0	0	1	9	3	0	0	0	0	4	167

right	2372
wrong	295
accuracy	0.889
macro-F1	0.893

(d) Confusion matrix on Jetson Nano with NAS-M TPU model

EfficientNet-B0 with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	64	3	1	1	0	1	1	0	0	0	0
food_bowl	0	170	0	0	0	0	0	1	0	0	0
food_box	4	1	221	0	0	1	0	2	1	1	0
glas_bottle	0	0	0	210	3	0	2	0	0	0	0
metal_can	0	0	0	17	245	0	1	0	0	0	0
plastic_bag	0	0	0	2	0	241	0	2	1	1	0
plastic_bottle	2	1	0	41	1	0	390	8	0	0	1
plastic_cup	1	6	0	4	1	2	1	347	0	0	0
plastic_cutlery	0	1	0	1	0	0	0	0	243	0	0
snack_wrap	0	0	2	0	1	0	1	0	0	222	4
tetrapack	0	0	1	1	0	0	0	0	0	0	186

right	2539
wrong	128
accuracy	0.952
macro-F1	0.951

(e) Confusion matrix on Jetson Nano with EffNet GPU model

EfficientNet-B0 with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	2	3	5	1	1	2	2	1	1	1
food_bowl	0	169	1	0	0	0	0	3	0	0	0
food_box	2	1	216	0	0	0	0	0	0	0	0
glas_bottle	0	0	0	190	2	0	8	0	0	0	0
metal_can	0	1	0	11	248	0	0	1	0	0	2
plastic_bag	1	0	1	4	0	243	0	1	0	1	0
plastic_bottle	0	3	3	54	0	1	386	12	0	0	0
plastic_cup	0	5	0	2	0	0	0	340	1	0	0
plastic_cutlery	0	1	1	3	0	0	0	0	243	0	0
snack_wrap	1	0	0	1	0	0	0	1	0	222	1
tetrapack	0	0	0	7	0	0	0	0	0	0	187

right	2511
wrong	156
accuracy	0.942
macro-F1	0.939

(f) Confusion matrix on Jetson Nano with EffNet TPU model

Figure 11.8: Confustion matrix from different CNN-Models on Jetson Nano

11.7 Precision and Recall

The following section discusses the precision, recall and F1 score obtained from the confusion matrices. The precision is a measure of how often the prediction of one class was correct. The recall describes how often a class was predicted correctly. The F1-Score is the harmonic mean value of these two key ratios.

In the matrix 11.9a the precision values of the 6 Convolutional Neuronal Networks can be viewed. The background class values of the TPU are noticeable. Compared to the other values, these are relatively low with MNV2 with 0.680, NAS-M with 0.798 and EffNet with 0.779. Furthermore, the NAS-M model has a very low precision for the plastic bottles, whereas the MNV2 and EffNet models also have a precision below 0.9. The values for the recall are listed in the matrix 11.9. Here the negative classification of the glass bottles is particularly apparent. The GPU models provide with 0.773 for the MNV2, 0.711 for the NAS-M and 0.758 for the EffNet a better recall value than the TPU models with 0.639 for the MNV2, 0.473 for the NAS-M and 0.686 for the EffNet. The F1 score values in the matrix 11.10 represent the harmonic mean of precision and recall. Here you can observe a low value of 0.786 for the background class of the TPU trained MNV2. For the glass bottle class there are the TPU models with 0.748 MNV2, 0.618 NAS-M and 0.797 EffNet and the GPU trained NAS-M with 0.780.

The following statements can be derived from precision, recall and F1 values. TPU trained models have a tendency to declare more classes as background than GPU trained models. In general the problem with the classification of glass bottles is confirmed. The model structure of the NAS-M shows more weaknesses in filtering the different features between glass and plastic bottles.

Precision						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.851	0.680	0.900	0.798	0.901	0.779
food_bowl	0.987	0.928	0.960	0.988	0.994	0.977
food_box	0.951	0.981	0.978	1.000	0.957	0.986
glas_bottle	0.930	0.903	0.864	0.891	0.977	0.950
metal_can	0.956	0.934	0.952	0.936	0.932	0.943
plastic_bag	0.979	0.930	0.992	0.964	0.976	0.968
plastic_bottle	0.851	0.850	0.788	0.693	0.878	0.841
plastic_cup	0.937	0.979	0.891	0.925	0.959	0.977
plastic_cutlery	0.946	0.980	0.988	0.946	0.992	0.980
snack_wrap	0.991	0.946	0.976	0.964	0.965	0.982
tetrapack	0.963	0.813	0.910	0.908	0.989	0.964

(a) Confusion matrix on Jetson Nano with MNV2 GPU model

Recall						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.887	0.930	0.887	0.944	0.901	0.944
food_bowl	0.857	0.918	0.934	0.929	0.934	0.929
food_box	0.956	0.942	0.969	0.862	0.982	0.960
glas_bottle	0.773	0.639	0.711	0.473	0.758	0.686
metal_can	0.952	0.964	0.952	0.936	0.976	0.988
plastic_bag	0.971	0.976	0.967	0.971	0.984	0.992
plastic_bottle	0.977	0.947	0.894	0.962	0.985	0.975
plastic_cup	0.944	0.903	0.928	0.922	0.964	0.944
plastic_cutlery	0.996	0.976	0.988	0.996	0.992	0.992
snack_wrap	0.973	0.933	0.911	0.955	0.991	0.991
tetrapack	0.958	0.953	0.958	0.874	0.974	0.979

(b) Confusion matrix on Jetson Nano with MNV2 TPU model

Figure 11.9: Confustion matrix from different CNN-Models on Jetson Nano

F1-value

F1						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.869	0.786	0.894	0.865	0.901	0.854
food_bowl	0.918	0.923	0.947	0.958	0.963	0.952
food_box	0.953	0.961	0.973	0.926	0.969	0.973
glas_bottle	0.844	0.748	0.780	0.618	0.854	0.797
metal_can	0.954	0.949	0.952	0.936	0.953	0.965
plastic_bag	0.975	0.952	0.979	0.967	0.980	0.980
plastic_bottle	0.910	0.896	0.838	0.805	0.929	0.903
plastic_cup	0.941	0.939	0.909	0.924	0.961	0.960
plastic_cutlery	0.970	0.978	0.988	0.970	0.992	0.986
snack_wrap	0.982	0.939	0.942	0.960	0.978	0.987
tetrapack	0.961	0.877	0.934	0.891	0.982	0.971
macro-F1	0.934	0.904	0.921	0.893	0.951	0.939

Figure 11.10: Average efficiency with different CNN-Models on Jetson Nano

12 Experiment Result of Intel Neuronal Compute Stick 2

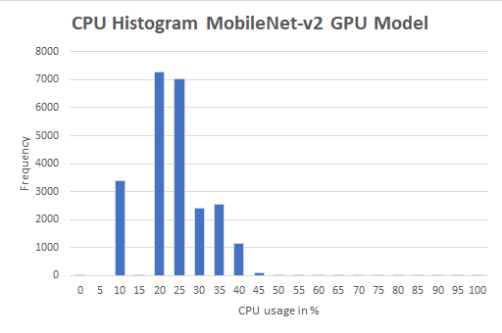
In this chapter the result of the experiment on the Intel Neuronal Compute Stick 2 is evaluated. The Neuronal Compute Stick 2 uses a Raspberry Pi 4 with 2GB of memory in this setting. The result includes the CPU and memory usage, the energy consumption, the inference time as well as a Confusions Matrix for each model, which determines the accuracy, macro-F1 and for each class the precision, recall and F1 score. The CNN models MobileNet-v2 and NASNet-Mobile in a GPU and TPU trained option each are used in this experiment. The EfficientNet-B0 is not yet supported by the Openvino framework, so it can not be executed on the VPU.The interaction with the host CPU takes place via the USB3 interface.

12.1 CPU Workload

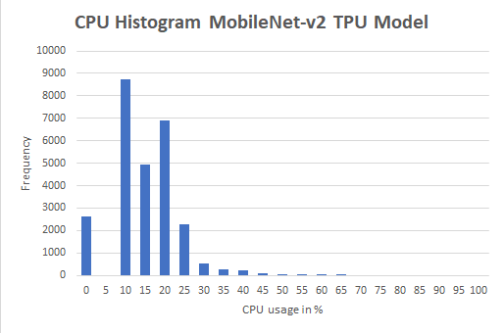
The following subchapter deals with the load of the host CPU during the inference with the 2 model architectures MNV2 and the NAS-M. The host CPU is an ARM v8 quad-core Cortex-A72 64-bit SoC with 1.5GHz.

In the figure 12.1 the measured values of the CPU workload per inference are plotted as a histogram. For the most part, the load of the models varies between 0% and 40%. The figure 12.2 shows the average values of the CPU utilization. It can be seen that the TPU trained models with 13% each load the host CPU less than the two GPU trained models with 22% for the MNV2 and 17% for the NAS-M.

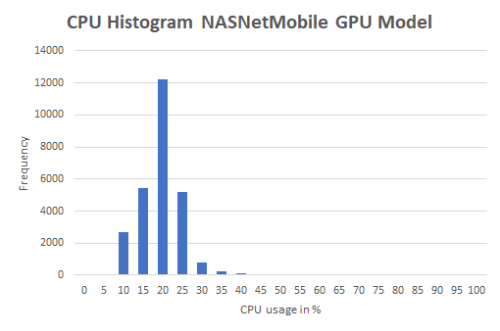
It shows that the host CPU, as with the Coral USB Accelerator, is not very heavily loaded by the USB accelerator. The models trained with the cloud TPU load the CPU even less than the cloud GPU trained models. In general, the host CPU does not have a great impact on the performance of the NCS2.



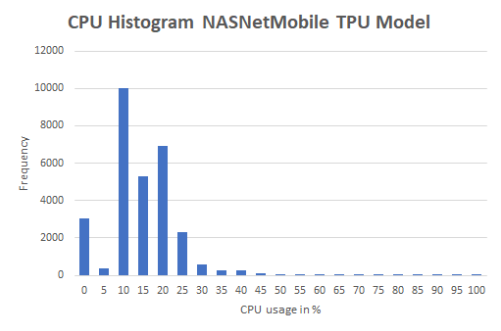
(a) Histogram of CPU workload on Neuronal Compute Stick 2 with MNV2 GPU model



(b) Histogram of CPU workload on Neuronal Compute Stick 2 with MNV2 TPU model



(c) Histogram of CPU workload on Coral USB Accelerator with NAS-M GPU model



(d) Histogram of CPU workload on Coral USB Accelerator with NAS-M TPU model

Figure 12.1: CPU workload histograms on Neuronal Compute Stick 2

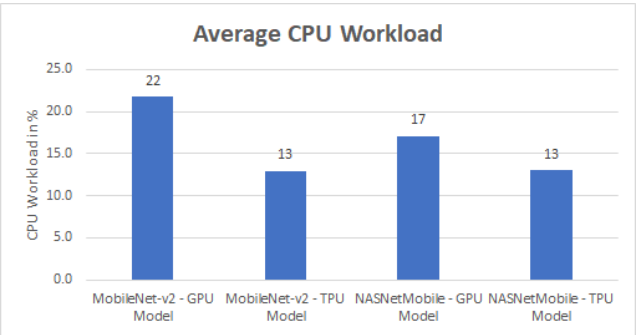


Figure 12.2: Average CPU Workload with different CNN-Models on Neuronal Compute Stick 2

12.2 Memory Workload

The following section shows the memory usage during the inference with the MNV2 and NAS-M CNNs discussed in their GPU and TPU trained version. In this case, the Raspberry Pi 4 has a 2GB LPDDR4-3200 SDRAM.

The bar graph 12.3 indicates the average memory usage during the inference in percent. The NAS-M in the GPU trained version stands out with a very low value of 19%. While the other models with 35% on the MNV2 GPU model and 44% on the TPU model or the TPU trained NAS-M with 38% are more or less in the same range.

The results demonstrate that the TPU trained models use more memory than the GPU trained models. Furthermore, it shows that the MNV2 needs more memory than the NAS-M models. It is expected that the NASNet-Mobile models with their larger structure has a higher memory consumption than the MobileNet-v2 with the lighter structure. The Neuronal Compute Stick 2 has, in contrast to the Coral USB Accelerator, a large 4GB internal DRAM memory, so the 2GB memory of the host CPU should not have a big impact. This suggests that the memory of the host CPU is occupied by other tasks of the operating system or the Openvino framework.

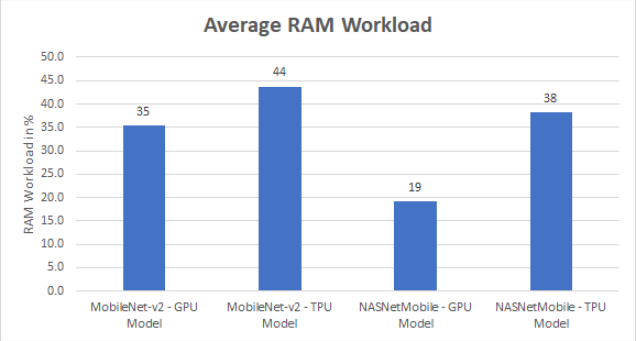


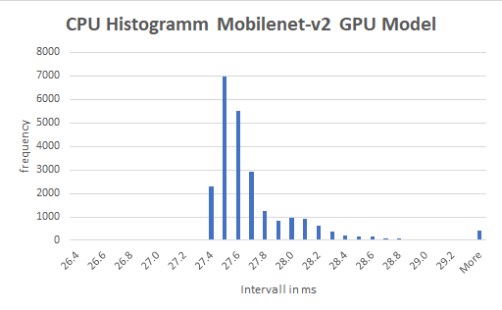
Figure 12.3: Average memory Workload with different CNN-Models on Neuronal Compute Stick 2

12.3 Inference Time

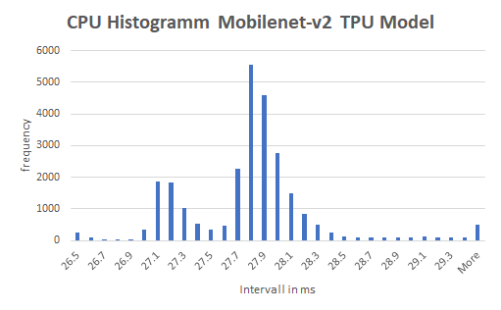
The following subchapter covers the inference time of the GPU and TPU trained MNV2 and NAS-M Convolutional Neuronal Networks on the Neuronal Compute Stick 2.

In the figure 12.4 the measured times of the 2 models MobileNet-v2, NASNet-Mobile with the respective GPU and TPU trained version are plotted as histogram. The MNV2 models operate according to the histograms 12.4a, 12.4b are in a range of 26ms to 29ms per classification. The time values measured for the two NAS-M 12.4c and 12.4 models are in a range from 77ms to 81ms per inference.The TPU trained models have a wider distribution than the GPU trained models. The diagram 12.5 represents the mean inference time of the 4 models. Obviously the MNV2 with 27.7ms (as GPU version) and 27.8ms (as TPU version) needs much less time than the NAS-M networks with 79.1ms (as GPU version) and 80.2ms (as TPU version). There is no significant difference between the GPU and TPU trained models at MNV2. With the NAS-M model, the TPU-trained model needs on average 1ms more for a prediction.

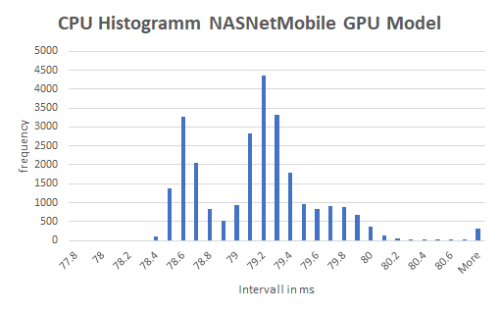
From the results of the histogram one can see that the MobileNet-v2 predicts much faster than the NASNet-Mobile. This is understandable because of the complexity and parameter differences between models. The determined average inference time of 28ms of the MNV2 corresponds to the result of 24ms from the benchmark test of the Mobilenet-v1 Libutti et al. 2020. Furthermore it can be seen that the inference time of the TPU trained models varies more than the GPU trained models.



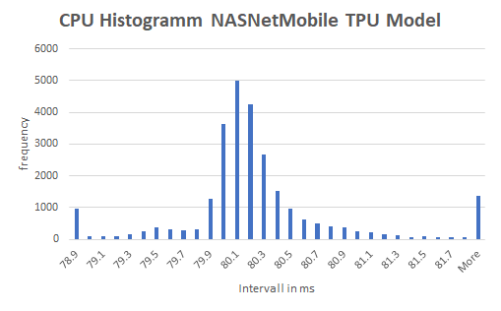
(a) Histogram of inference time on Neuronal Compute Stick 2 with MNV2 GPU model



(b) Histogram of inference time on Neuronal Compute Stick 2 with MNV2 TPU model



(c) Histogram of inference time on Neuronal Compute Stick 2 with NAS-M GPU model



(d) Histogram of inference time on Neuronal Compute Stick 2 with NAS-M TPU model

Figure 12.4: Inference time histograms on Neuronal Compute Stick 2

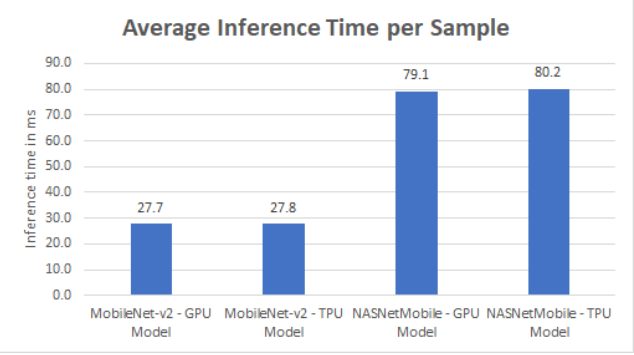


Figure 12.5: Average inference time with different CNN-Models on Neuronal Compute Stick 2

12.4 Power Cosumption

In this section the energy consumption of the Neuronal Compute Stick 2 during the classification with the MobileNet-v2 and NASNet-Mobile models is discussed. Both models use the cloud-GPU and cloud-TPU options. Unlike the other models, the NCS2 does not have a power reduction setting.

In the following bar chart 12.6 the mean values of the energy consumption of the 4 Convolutional Neuronal Networks are shown. All models are in the range of 5.5W, the MNV2 needs 5.6W whereas the 0.1W are negligible. In idle mode the host CPU with the NCS2 needs 3.53W on average.

With the Neuronal Compute Stick 2 there is no influence of the model architecture on the power consumption. There is also no difference in power consumption between the GPU and TPU trained models. All models in this experiment required 5.5W during inference, while in sleep mode 3.5W is consumed. This results in an average consumption of 2W at inference, which is in conformance with the result of the benchmark test of the Mobilenet-v1. Libutti et al. 2020

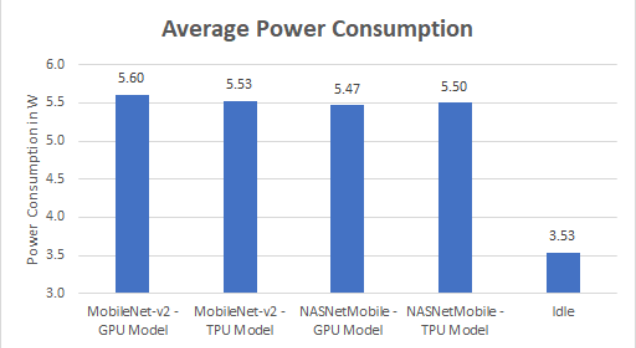


Figure 12.6: Average power consumption with different CNN-Models on Neuronal Compute Stick 2

12.5 Efficiency

In this section the efficiency of the Neuronal Compute Stick 2 is explained for the models MobileNet-v2 and NASNet-Mobile.

The bar chart 12.7 indicates the mean efficiency for the following models in mW: MNV2 GPU trained, MNV2 TPU trained, NAS-M GPU trained and NAS-M TPU trained. The two MNV2 CNNs require 155mW or 152mW with the TPU version for a prediction. The NAS-M models have an average consumption of 437mW with the GPU version or 441mW with the TPU version for a classification.

It turns out that the MobileNet-v2 models in both versions require about one third of the energy for a prediction than the NASNet-Mobile models. There is no significant difference between a GPU and TPU trained model.

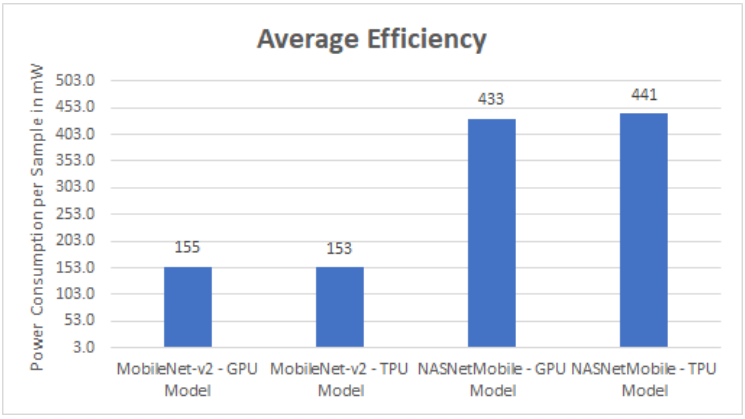


Figure 12.7: Average efficiency with different CNN-Models on Neuronal Compute Stick 2

12.6 Confusion Matrix

In this subchapter the created confusion matrices of the Convolutional Neuronal Networks performed in the experiment are explained. The models are the MobileNet-v2 in a cloud GPU and TPU trained version and the NASNet-Mobile in the GPU and TPU trained version. When the matrix is generated, all 2667 images of the test data set are classified with each model on the Neuronal Compute Stick 2. The predicted result determines the row of the confusions matrix while the real class specifies the column. From the confusion matrix the precision, the macro-F1 and for each class the precision, recall and F1 score are estimated.

In the figure 12.8 the confusion matrices of the 4 neural networks MobileNet-v2 trained with GPU and TPU and the NASNet-Mobile trained with GPU and TPU are shown. The quality of the Convolutional Neuronal Network is determined with the accuracy and the macro-F1. The accuracy is the ratio of all correctly classified samples to all incorrectly classified samples. The macro-F1 is the mean of the harmonic mean of precision and recall of all classes and has a higher power if the number of test data per class is unequal. For the MNV2 models the accuracy is 0.933 for the GPU model and 0.911 for the TPU model while 0.931 for the GPU model and 0.854 for the TPU model for the macro-F1.For the NASNet-Mobile the following values for the accuracy result: 0.846 with the GPU trained model and 0.780 with the TPU trained model plus for the macro-F1 0.902 with the GPU trained model and 0.772 with the TPU trained model.

The determined accuracy and macro-F1 of the models clearly shows that the TPU trained models classify the classes worse than the GPU trained models. Furthermore, it is obvious that the MNV2 models generally give a better result than the NAS-M models. Noticeable is the glass bottle and plastic cup class on the NAS-M GPU model as well as the glass bottle and plastic bottle class on the TPU model. The openvino optimized NAS-M models give a poor prediction for these classes.

MobileNet with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	71	2	4	1	0	4	2	5	3	2	1
food_bowl	0	166	0	0	0	0	0	1	0	0	0
food_box	0	1	213	0	0	0	0	1	1	1	0
glas_bottle	0	0	0	207	2	0	4	0	0	0	0
metal_can	0	0	0	20	247	0	2	0	1	2	0
plastic_bag	0	0	1	1	0	237	0	1	0	0	0
plastic_bottle	0	2	0	33	1	0	380	15	0	0	3
plastic_cup	0	11	1	10	0	4	6	337	2	1	0
plastic_cutlery	0	0	2	0	0	0	0	0	238	0	0
snack_wrap	0	0	4	0	0	0	0	0	0	206	1
tetrapack	0	0	0	5	1	0	2	0	0	12	186

right	2488
wrong	179
accuracy	0.933
macro-F1	0.931

(a) Confusion matrix on Neuronal Compute Stick 2 with MNV2 GPU model

MobileNet with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	66	5	4	5	2	4	1	7	3	2	2
food_bowl	1	168	1	0	0	0	0	9	2	0	0
food_box	2	0	210	0	0	0	0	0	1	1	0
glas_bottle	0	0	0	175	2	0	14	2	0	0	1
metal_can	0	2	0	11	243	0	0	1	0	0	3
plastic_bag	1	1	4	3	0	239	0	7	0	1	0
plastic_bottle	1	3	0	47	0	1	376	10	1	1	2
plastic_cup	0	3	0	4	0	0	0	324	0	0	0
plastic_cutlery	0	0	2	1	0	0	1	0	238	0	0
snack_wrap	0	0	4	6	1	1	0	0	0	209	1
tetrapack	0	0	0	25	3	0	4	0	0	10	182

right	2430
wrong	237
accuracy	0.911
macro-F1	0.854

(b) Confusion matrix on Neuronal Compute Stick 2 with MNV2 TPU model

NASNet with GPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	56	1	3	2	0	8	1	2	2	0	1
food_bowl	2	167	2	3	0	0	1	4	2	0	0
food_box	0	0	208	0	0	2	0	0	0	1	0
glas_bottle	6	0	0	179	19	1	29	0	0	0	1
metal_can	0	0	0	10	207	0	0	0	0	0	0
plastic_bag	0	0	2	0	0	203	1	1	0	1	0
plastic_bottle	4	3	2	54	8	24	346	76	0	3	4
plastic_cup	0	11	2	14	1	4	12	275	3	1	0
plastic_cutlery	1	0	3	3	2	2	2	2	238	1	1
snack_wrap	2	0	1	0	0	1	0	0	0	196	3
tetrapack	0	0	2	12	14	0	4	0	0	21	181

right	2256
wrong	411
accuracy	0.846
macro-F1	0.902

(c) Confusion matrix on Neuronal Compute Stick 2 with NAS-M GPU model

NASNet with TPU Model

	background	food_bowl	food_box	glas_bottle	metal_can	plastic_bag	plastic_bottle	plastic_cup	plastic_cutlery	snack_wrap	tetrapack
background	67	6	36	7	5	7	5	14	14	4	4
food_bowl	1	161	4	3	0	2	0	5	1	0	0
food_box	0	0	165	0	0	0	0	0	0	0	0
glas_bottle	1	0	0	63	1	0	16	0	0	0	1
metal_can	0	0	0	12	207	0	0	0	0	3	0
plastic_bag	0	0	3	3	0	224	3	10	0	1	0
plastic_bottle	0	3	0	137	28	10	368	70	0	7	37
plastic_cup	1	9	3	6	3	0	3	255	0	1	0
plastic_cutlery	0	0	7	3	0	1	1	5	229	1	0
snack_wrap	1	0	5	1	0	0	0	0	0	194	3
tetrapack	0	3	2	42	7	1	0	1	1	13	146

right	2079
wrong	588
accuracy	0.780
macro-F1	0.772

(d) Confusion matrix on Neuronal Compute Stick 2 with NAS-M TPU model

Figure 12.8: Confustion matrix from different CNN-Models on Neuronal Compute Stick 2

12.7 Precision, Recall and F1-value

The following subchapter deals with the precision, recall and F1 score determined for each class. These characteristics are determined from the confusion matrices of the models MobileNet-v2 and NASNet-Mobile. The precision describes how often the prediction of a class was correct, while the recall indicates the ratio of the incorrectly determined samples of a class to their total number. The F1-score is the harmonic mean of precision and recall of a class.

In the figure 12.9a the precision values of the classes are visible. The background class of the MNV2 with 0.747 (GPU) and 0.653 (TPU) as well as the NAS-M with 0.737 (GPU) and 0.396 (TPU) have a low precision. Also problematic are the precision values of the NAS-M models for the glass bottle class with 0.762 (GPU), 0.768 (TPU), the plastic bottle class with 0.660 (GPU), 558 (TPU) and the tetrapack class with 0.774 (GPU), 0.676 (TPU). The recall of the respective classes and CNN models are shown in the figure 12.9. As expected, the glass bottle class shows a bad result with 0.747 for the GPU and 0.632 for the TPU trained model of the MNV2 as well as with 0.646 for the GPU trained and 0.227 for the TPU trained model of the NAS-M. Other notable low values are the background with 0.789 and the plastic cup with 0.764 on the GPU model of the NAS-M as well as the plastic cup with 0.708 and the tetrapack with 0.764 on the TPU trained model of the NAS-M.The matrix in 12.10 illustrates the F1 score of classes and models. The values of the background class are low with 0.767 for the TPU model of the MNV2 as well as 0.762 for the GPU and 0.558 for the TPU model of the NAS-M. Also for the glass bottle class, the results are small with 0.743 (TPU) MNV2, 0.699 (GPU) NAS-M and 0.351 (TPU) NAS-M. For the NASNet-Mobile model, the plastic bottle class with 0.752 for the GPU and 0.796 for the TPU model as well as the tetrapack class with 0.717 for the TPU model are still in a critical range.

The results of the precision, recall and F1 scores show that the model deteriorates significantly after the Openvino optimization. This is clearly evident in the background class where many samples were incorrectly classified as background. The same applies to the NASNet-Mobile model and the plastic bottles and tetrapack. In general the glass bottles are difficult to predict, but the NAS-M TPU model is not able to detect these objects. The NAS-M models also perform poorly when detecting plastic cups and tetrapack.

Precision and Recall

Precision						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.747	0.653	0.737	0.396	None	None
food_bowl	0.994	0.928	0.923	0.910	None	None
food_box	0.982	0.981	0.986	1.000	None	None
glas_bottle	0.972	0.902	0.762	0.768	None	None
metal_can	0.908	0.935	0.954	0.932	None	None
plastic_bag	0.988	0.934	0.976	0.918	None	None
plastic_bottle	0.876	0.851	0.660	0.558	None	None
plastic_cup	0.906	0.979	0.851	0.907	None	None
plastic_cutlery	0.992	0.983	0.933	0.927	None	None
snack_wrap	0.976	0.941	0.966	0.951	None	None
tetrapack	0.903	0.813	0.774	0.676	None	None

(a) Confusion matrix on Neuronal Compute Stick 2 with MNV2 GPU model

Recall						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	1.000	0.930	0.789	0.944	None	None
food_bowl	0.912	0.923	0.918	0.885	None	None
food_box	0.947	0.933	0.924	0.733	None	None
glas_bottle	0.747	0.632	0.646	0.227	None	None
metal_can	0.984	0.968	0.825	0.825	None	None
plastic_bag	0.967	0.976	0.829	0.914	None	None
plastic_bottle	0.960	0.949	0.874	0.929	None	None
plastic_cup	0.936	0.900	0.764	0.708	None	None
plastic_cutlery	0.971	0.971	0.971	0.935	None	None
snack_wrap	0.920	0.933	0.875	0.866	None	None
tetrapack	0.974	0.953	0.948	0.764	None	None

(b) Confusion matrix on Neuronal Compute Stick 2 with MNV2 TPU model

Figure 12.9: Confustion matrix from different CNN-Models on Neuronal Compute Stick 2

F1-value

F1						
	MobileNet-v2		NASNet-Mobile		EfficientNet-B0	
	GPU	TPU	GPU	TPU	GPU	TPU
background	0.855	0.767	0.762	0.558	None	None
food_bowl	0.951	0.926	0.920	0.897	None	None
food_box	0.964	0.957	0.954	0.846	None	None
glas_bottle	0.845	0.743	0.699	0.351	None	None
metal_can	0.945	0.951	0.885	0.875	None	None
plastic_bag	0.977	0.954	0.896	0.916	None	None
plastic_bottle	0.916	0.897	0.752	0.697	None	None
plastic_cup	0.921	0.938	0.805	0.796	None	None
plastic_cutlery	0.981	0.977	0.952	0.931	None	None
snack_wrap	0.947	0.937	0.918	0.907	None	None
tetrapack	0.937	0.877	0.852	0.717	None	None
macro-F1	0.931	0.902	0.854	0.772	None	None

Figure 12.10: Average efficiency with different CNN-Models on Neuronal Compute Stick 2

13 Conclusion of the Evaluation

In this chapter the results of the evaluation of the AI-edge devices Coral USB Accelerator, Jetson Nano and Neuronal Compute Stick 2 are discussed. It is determined which combinations of Convolutional Neuronal Network and end device are used for the prototype.

In the table 13.1 are the results of CPU usage, memory consumption, inference time, power consumption, efficiency and the quality of CNN as accuracy and macro-F1 for each model and for all 3 AI accelerators. The NCS2 with the NASNet-Mobile trained with the cloud TPU has the lowest CPU usage with 13%. In this case the CPU usage plays a minor role, since no model or device reaches a critical value.
In this experiment the CUA with values between 10% (400MB) and 11% (440MB) needs the least memory. Especially the Nano has a very high memory usage, which can lead to problems during the execution of the prototype, because memory is still needed for video streaming and displaying the camera views on the screen.
In the inference time, the Coral Edge TPU is clearly outperforming the other AI accelerators. The best result of 5.7ms is achieved at the maximum operation frequency with the MNV2 with the GPU and TPU trained version. For the real life application a low inference time means that more images can be classified in the same time period.
In terms of power consumption, the USB accelerators are the same, only the CUA needs less than average with the NAS-M at the standard operating frequency. The Jetson Nano consumes significantly more power than the other two edge devices. With the efficiency the advantage of the embedded Tensor Processing Unit is clearly shown. With 31mW for the MNV2 and 79mW for the NAS-M per classification of an image, the computing unit requires significantly less energy than the other technologies.
With the cloud GPU trained MobileNet-v2 model a great accuracy as well as macro-F1 is achieved on all devices. The highest accuracy with 95.2% and a macro-F1 of 95-1 is achieved with the EfficientNet-B0 on the Jetson Nano.The EffNet is not yet supported by the Coral and Openvino platforms at the time of this writing. Since the EffNet is an extension of the MNV2 it is assumed that the accuracy and macro-F1 are in the same range for the edge TPU and VPU.

As the best solution for the classification of waste objects the combination of the Coral USB Accelerator and the MobileNet-v2 is assumed. To get a better accuracy the GPU trained version is used.

Models	CPU Workload	MEM Workload	Inference Time	Power Consumption	Efficiency	Accuracy	Macro-F1
Coral on STD - MNV2 GPU trained	18%	392MB	7.1ms	5.43W	39mW	93.5%	93.3%
Coral on STD - MNV2 TPU trained	21%	400MB	7.2ms	5.50W	40mW	90.7%	89.9%
Coral on STD - NASMobile GPU trained	16%	424MB	21ms	5.17W	108mW	90.5%	91.4%
Coral on STD - NASMobile TPU trained	16%	424MB	21ms	5.17W	109mW	87.6%	88.1%
Coral on MAX - MNV2 GPU trained	20%	412MB	5.7ms	5.51W	31mW	93.5%	93.3%
Coral on MAX - MNV2 TPU trained	20%	412MB	5.7ms	5.50W	31mW	90.7%	89.9%
Coral on MAX - NASMobile GPU trained	17%	424MB	14.3ms	5.51W	79mW	90.5%	91.4%
Coral on MAX - NASMobile TPU trained	17%	428MB	14.3ms	5.51W	79mW	87.6%	88.1%
Jetson Nano - MNV2 GPU trained	27%	3.36GB	17.6ms	7.52W	132mW	93.6%	93.4%
Jetson Nano - MNV2 TPU trained	27%	3.44GB	17.6ms	7.68W	135mW	91.2%	90.4%
Jetson Nano - NASMobile GPUtrained	37%	3.68GB	47.2ms	8.76W	414mW	91.5%	92.1%
Jetson Nano - NASMobile TPU trained	38%	3.60GB	48.0ms	8.89W	430mW	88.9%	89.3%
Jetson Nano - EffNet GPUtrained	30%	3.92GB	35.0ms	8.51W	298mW	95.2%	95.1%
Jetson Nano - EffNet TPU trained	29%	3.68GB	35.5ms	8.42W	299mW	94.2%	93.9%
NCS2 - MNV2 GPU trained	22%	700MB	27.7ms	5.60W	155mW	93.3%	93.1%
NCS2 - MNV2 TPU trained	13%	880MB	27.8ms	5.53W	153mW	91.1%	85.4%
NCS2 - NASMobile GPU trained	17%	380MB	79.1ms	5.47W	433mW	84.6%	90.2%
NCS2 - NASMobile TPU trained	13%	760MB	80.2ms	5.50W	441mW	78.0%	77.2%

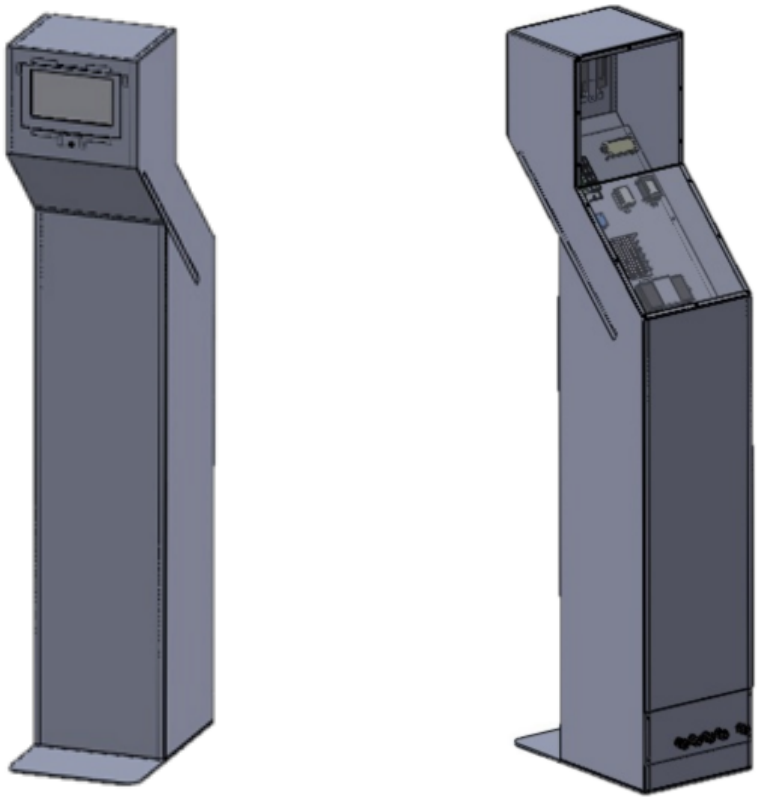
Table 13.1: Overview of the evaluation results of the Convolutional Neuronal Networks on the edge devices Coral USB Accelerator Jetson Nano Neuronal Compute Stick 2

14 Prototype

The following chapter covers the prototype for testing the concept under real conditions. Furthermore, the prototype was exhibited in cooperation with the Faculty of Environmental Engineering at the university fair Kasetfair2020. An opinion survey for the use of Artificial Intelligence in the field of recycling was realised and the handling of the machine and the performance of the CNN got be observed in general.

To test the concept of waste classification for suitability a simple system is created on a Raspberry Pi 4. As embedded AI accelerator the Coral USB Accelerator is used with the MobileNet-v2 as GPU trained version. This combination has proven to be the most suitable solution in terms of energy consumption, speed, efficiency and accuracy. A Raspberry PI camera V2.1 with 5MP over the CSI interface is used to record the video stream. The stream runs on the lowest resolution of 640x480 which allows a frame rate of 40-90 fps *6. Camera Hardware — Picamera 1.12 Documentation* 2020.To display the field of view of the camera and the currently detected class of the CNN a 7' inch TFT display is used via the mini HDMI interface. S3003 servo motors are used for the waste bin opening mechanism. For the mechanics, a 3D printer produces swivel arms and a clap adapted to the opening of the bins, illustrated in the CAD figure 14.2. This construction is attached to the head of the bin, which allows the buckets to be opened or closed with a 120° turn. A PCA9685 pulse width modulator (PWM) driver is used to control the servo motors. This driver is controlled via the I2C interface of the Raspberry Pi 4. The power supply is a Damper S-50-5 power supply unit which supplies 5V and 10A. In the illustration 14.1 the front and back of the terminal is illustrated for the prototype. The hardware components are located inside this terminal, see 14.1b. The camera is attached to the diagonal edge below the screen, see front view 14.1a. The camera is thus directed towards the ground, making the background more constant than when looking straight ahead. This helps to avoid spontaneous, unwanted classifications.

The control logic is visible in the flow chart 14.3. The process starts with reading of a new frame from the Raspberry PI camera. The acquired frame is scaled down from 640x480 to 224x224 in preprocessing and converted to a numpy array to match the format of the input tensor of the MNV2. The CNN classifies the processed image into one of 11 classes. The first step of the process checks if the reliability of the prediction falls below the threshold of 0.6. If it does, the result is ignored and the next frame is read. If it is exceeded, it is checked whether it is a background. A detected background means that there is no object and therefore the next frame is read in. If the result is one of the 10 waste categories the initialization of the object is started by saving the current time stamp and the predicted class. If the result changes within 1 second, the initialization time and the class is reassigned. If the result remains the same, the corresponding servomotor for this predicted category is triggered. The servo motor makes a 120° turn to bring the waste bin flap into an open position and a 5s timer is started. During this opening time no new frames are read. After the 5s opening time has passed, the flap is moved back to the closed position and all variables are initialized to 0 again. The process starts again from the beginning.



(a) Front view of the prototype terminal (b) Back view of the prototype terminal

Figure 14.1: Mechanical construnction of the prototype terminal

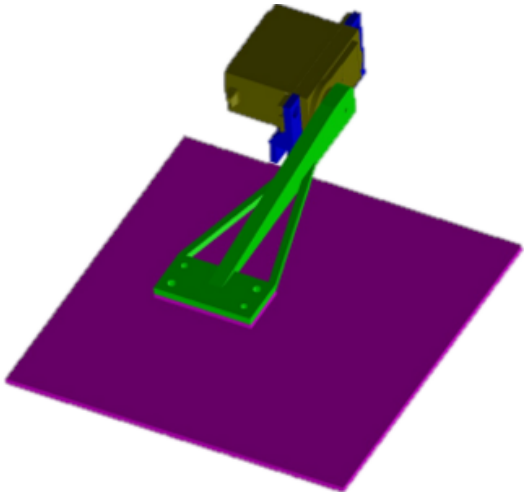


Figure 14.2: CAD model of the bin open mechanic with a S3003 servomotor for the prototype

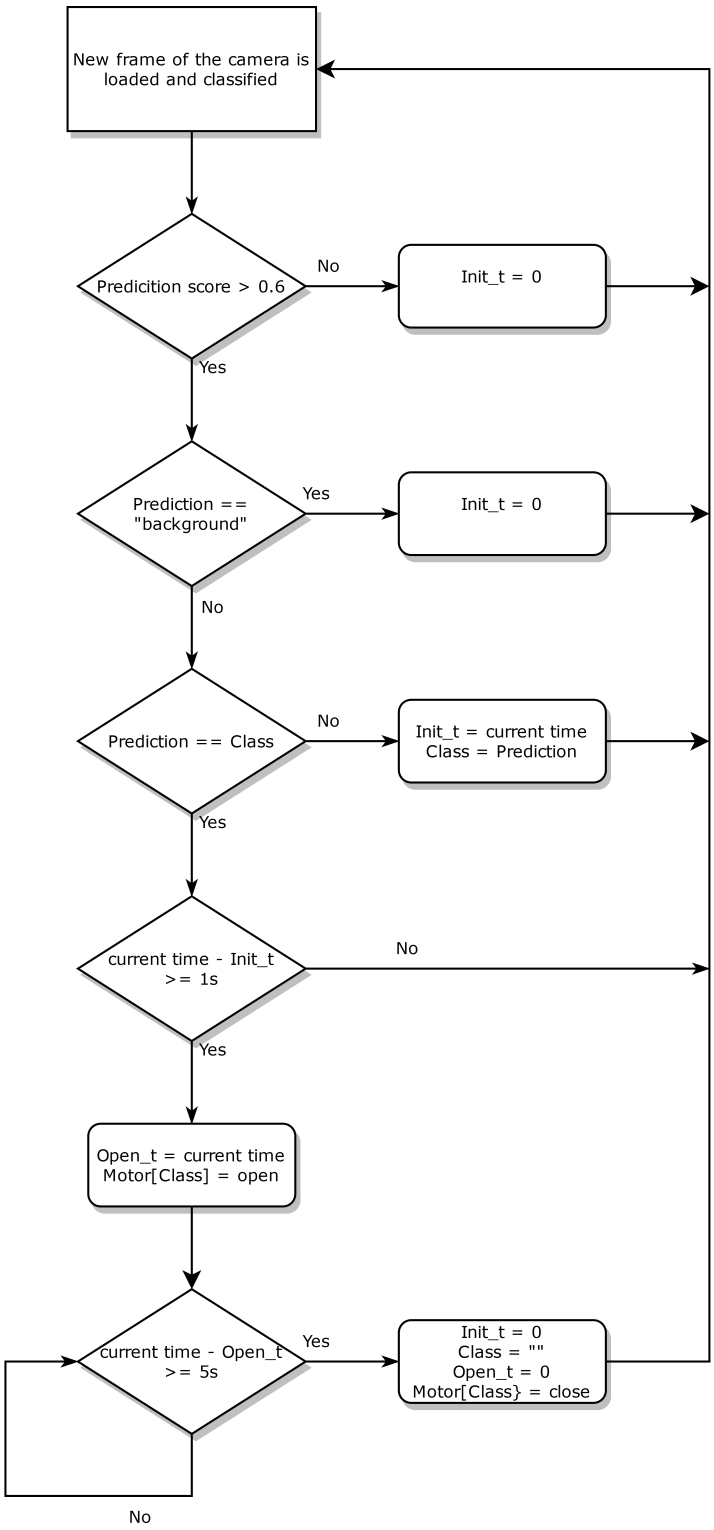


Figure 14.3: Flowchart of the opening logic of the prototype

Usage at Kasetfair2020

The following subchapter deals with the experience gained with the prototype under real conditions at the KasetFair2020. Furthermore, the results of the opinion survey made in collaboration with Assist.Prof.Cheema Soralump are presented in this section.

The prototype was exhibited at the KasetFair2020 fair from 31 January 2020 to 8 February 2020. The aim was to gain experience about the behaviour of the machine and the users and to collect real life pictures to improve the Convolutional Neuronal Network. It turns out that most users were overwhelmed with the first application of the machine, so personal support was required. Therefore a measurement of the accuracy and speed was omitted.

The following problems occurred in the application of the prototype:

- Reference to camera position is not sufficient to define the camera's field of view.
- Classification sometimes too slow due to classes changing and the resulting reset of the initialization time.
- Not visible that the trash bin has opened after the classification.

From a technical point of view, the following observations were made:

- Pay more attention to lighting conditions
- Incorrect classification due to incorrect positioning of the object
- Self-locking of the servo motors is not sufficient to keep the flap closed

The knowledge gained results in improvements in the display of the camera's field of view. Furthermore, the process is not ideal and must be optimized to reduce the time needed to classify the object. The opening mechanism has to be designed for a higher load by the user. It also turns out that the camera view in outdoor use must be better shielded from sunlight. The position of the sun and reflections influence the result of the classification too much. Furthermore, one camera is not sufficient to detect the objects in different positions, therefore additional cameras are necessary. The opening mechanics will quickly become defective if used incorrectly and should be redesigned to prevent incorrect use.

The illustration 14.4 shows the results of the opinion research. The survey is to be observed critically as the answers are rather too positive for reasons of politeness and friendliness. Basically it can be seen that young adults with academic discharge are aware of the importance of recycling. The knowledge about correct recycling is generally good, although there is an interest in the support of recycling through artificial intelligence.

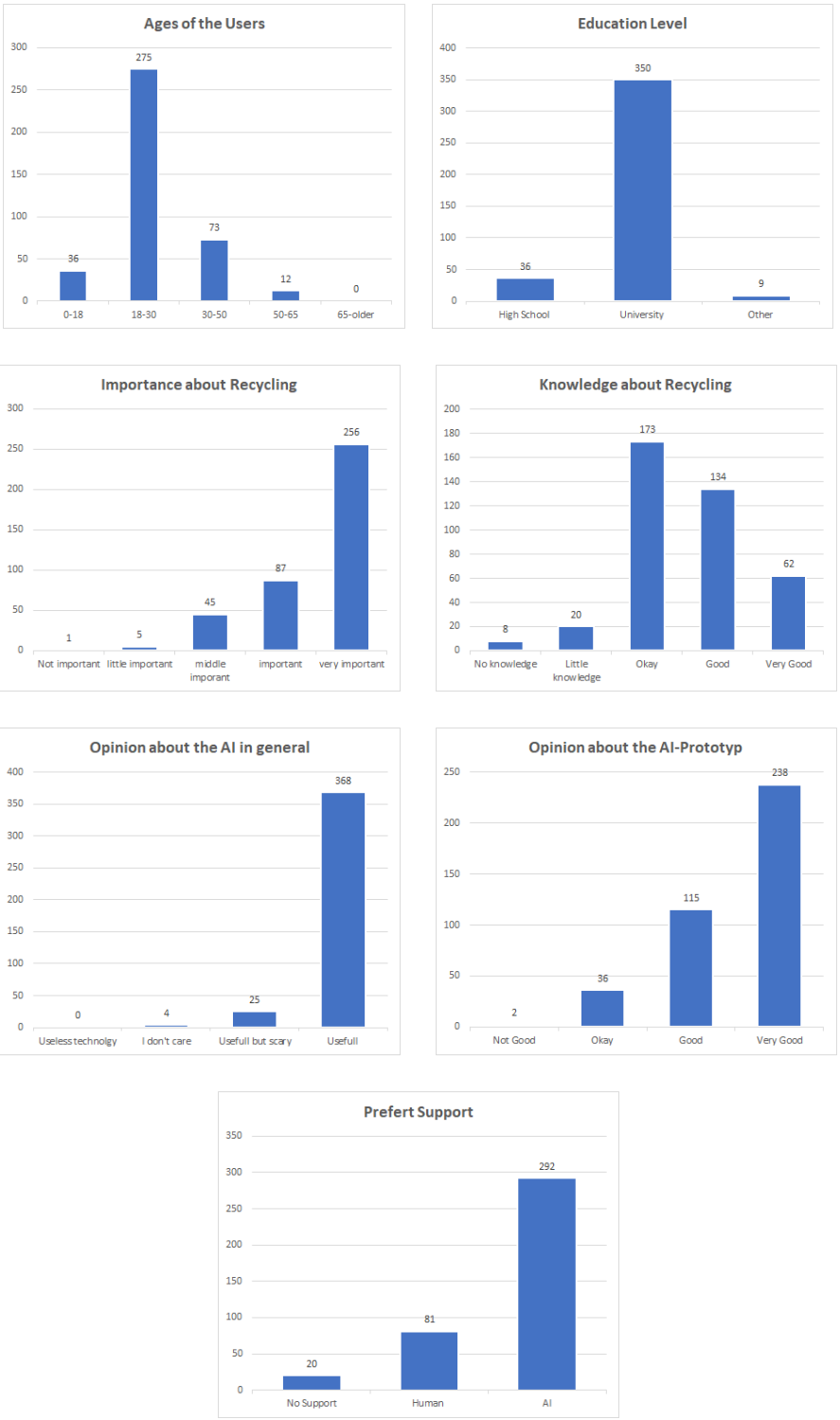


Figure 14.4: Result of the opinion survey at Kasetfair2020

15 Conclusion

In this thesis a concept is presented which classifies waste objects with the use of a Convolutional Neuronal Network and opens the container for this waste category. Together with the Faculty of Environmental Engineering 10 categories of this food bowl, food box, glass bottle, metal can, plastic bag, plastic bottle, plastic cup, plastic cutlery, snackwrap and tetrapack have been defined. With this selection the majority of public waste is covered. Of these 10 classes plus the background class, a data set of 25,681 samples has been created. This data set is divided into 70% training data, 20% validation data and 10% test data. The concept foresees an outdoor use and therefore the focus is on energy consumption in order to simplify the use of solar cells. For this purpose, current AI accelerator devices specially developed for embedded use are purchased and tested for use. The hardware models are the Google Coral USB Accelerator which is equipped with an edge-Tensor Processing Unit developed for low performance. The Nvidia Jetson Nano in the developer board version which is the smallest version of the Jetson series with the Tegra Graphics Processing Unit. The third device is an Intel Neuronal Compute Stick 2 which uses a Myriad X Vision Processing Unit developed by Movidius. Furthermore, 3 different CNN architectures are trained and tested. To train and manipulate the neural networks the Tensorflow/Keras framework is used and executed on the Google Colaboratory Platform. This Keras platform offers a number of pre-trained CNN models, for the use in the concept 3 mobile versions are used. These models are the MobileNet-v2, the NASNet-Mobile and the EfficientNet-B0. These models are trained in 2 versions, once with the Google cloud GPU and once with the cloud TPU. It will be examined whether the technology has an impact on the performance or quality of a neural network. After training, the Keras models are tested for quality. The models classify the test data set of 2667 samples and with the result a confusion matrix is created from which the accuracy as well as the macro-F1 is determined. The result of this test already demonstrated that the GPU trained models have a higher accuracy and macro-F1 than the GPU trained models. The best result was the EfficientNet-B0, followed by the MobileNet-v2. All 3 models delivered a suitable accuracy for the concept. In order to run the models on the end devices, they must first be converted and optimized for the respective platform of the manufacturers. The models of the Coral USB Accelerator are first converted to a TF-lite format and then the parameters are optimized to a quantized 8-bit INT format. For the GPU of the Jetson Nano no conversion is necessary, for better performance the models are optimized for the TF-TensorRT format. The parameters of the model are brought to half precision 16-bit floating point. For the Neuronal

Compute Stick 2 the CNNs have to be optimized to the Openvino format. As with the Nano, the parameters are optimized to half precision and then converted to a Intermediate Representation format. Thereby, it turns out that the most current model, the EfficientNet-B0 cannot be converted to the CUA and NCS2. Since these platforms do not support all functions of the model. To determine the most suitable combination of edge-AI-accelerator and Convolutional Neuronal Network architecture, a verification is performed where all CNN models are executed on the 3 devices. The CPU and memory usage, inference time and current consumption during inference are measured. Furthermore, as with the Keras models, a confusions matrix is created to determine the quality of the models.The result of the tests shows that with the MobileNet-v2 on the edge-TPU of the Coral USB Accelerator the best values for the inference time and energy consumption are reached and therefore the best efficiency is achieved. Also the accuracy with 93.5% and the macro-F1 with 93.3% are within a reasonable range in this combination. The CUA generally gives the best result for all models (except the EffNet). In terms of accuracy and macro-F1, the Jetson Nano gives slightly better results than the CUA, the best result being are obtained with the EfficientNet-B0. However, the Jetson Nano requires significantly more energy than the other two AI on the edge devices. The Neuronal Compute Stick 2 is on the same level of consumption as the CUA but provides a much slower inference time and therefore, a reduced efficiency. Furthermore, only the MobileNet-v2 on the NCS2 provides a suitable result. From the result of the experiment the Coral USB Accelerator is determined as the most suitable hardware in combination with the GPU trained MobileNet-v2 for the concept. A prototype was developed to test the concept under real conditions. The prototype was tested at the KasetFair2020 fair. In addition, a survey is conducted with the Faculty of Environmental Engineering to get the opinion of the users about Artificial Intelligence in the field of supported recycling. The results of the test indicate that the concept works properly, although from a technical point of view, weaknesses in the classification due to different light influences as well as in the orientation of the waste object to the camera field of view have arisen. Furthermore, the user was not immediately aware of how the concept works, which shows that there must be a greater focus on ease of use. The survey shows that the main interest comes from young adults with an academic education, whereby the concept itself received a lot of support from all participants.

Future Work

This work demonstrates that the concept works and that there is interest. To develop a concept suitable for everyday use, performance and reliability must be improved. Therefore, a new architecture of a neural network could be designed, because the used models are developed for a classification of 1000 classes. The design of an own board for the use of AI accelerators has to be considered. Thus, the edge-TPU is also available as a system on a chip (SoC) as a module or via other interfaces like M.2 and PCIe, which allow a higher data transfer rate. From a performance point of view, using the Python environment is not ideal, so a switch to a C++ environment is required. Coral already offers C++ libraries for the edge-TPU and recommends them for optimized performance. Furthermore, the mechanical design of the opening system has to be reconsidered in order to withstand the application under real conditions and the resulting load. To achieve a better classification, more than one camera should be used to detect the outline of the waste object, even if it is not oriented correctly. Also an infrared camera could be implemented to distinguish glass from plastic or to detect ice in plastic cups. Interesting are also other concepts that are in the recycling area with the idea of AI on the edge. Similar concepts could be researched with the use of object recognition and segmentation. Further, more other applications could be found in the field of robotics as an additional security in the disposal of medical waste or as a cheaper solution for machine vision.

Bibliography

6. Camera Hardware — Picamera 1.12 Documentation (July 2020). https://picamera.readthedocs.io/en/release-1.12/fov.html (cit. on p. 91).

Albawi, Saad; Mohammed, Tareq Abed, and Al-Zawi, Saad (Aug. 2017): "Understanding of a Convolutional Neural Network". In: *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186 (cit. on p. 21).

Antonini, Mattia et al. (Nov. 2019): "Resource Characterisation of Personal-Scale Sensing Models on Edge Accelerators". In: *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. AIChallengeIoT'19. New York, NY, USA: Association for Computing Machinery, pp. 49–55. ISBN: 978-1-4503-7013-4. DOI: 10.1145/3363347.3363363 (cit. on pp. 13, 14, 45).

Baji, Toru (Mar. 2018): "Evolution of the GPU Device Widely Used in AI and Massive Parallel Processing". In: *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*, pp. 7–9. DOI: 10.1109/EDTM.2018.8421507 (cit. on pp. 10, 15).

Barry, Brendan; Brick, Cormac, et al. (Mar. 2015): "Always-on Vision Processing Unit for Mobile Applications". In: *IEEE Micro* 35.2, pp. 56–66. ISSN: 1937-4143. DOI: 10.1109/MM.2015.10 (cit. on pp. 10, 17, 18).

Barry, Brendan; Connor, Fergal, et al. (Feb. 2015): "Vector Processor". en. US20150046673A1 (cit. on p. 10).

Bircanoğlu, Cenk et al. (July 2018): "RecycleNet: Intelligent Waste Sorting Using Deep Neural Networks". In: *2018 Innovations in Intelligent Systems and Applications (INISTA)*, pp. 1–7. DOI: 10.1109/INISTA.2018.8466276 (cit. on p. 6).

Booklet on Thailand State of Pollution (2019). Pollution Control Department, Ministry of Natural Resources and Environment. ISBN: 978-616-316-511-4 (cit. on pp. 4, 11).

Cass, Stephen (May 2019): "Taking AI to the Edge: Google's TPU Now Comes in a Maker-Friendly Package". In: *IEEE Spectrum* 56.5, pp. 16–17. ISSN: 1939-9340. DOI: 10.1109/MSPEC.2019.8701189 (cit. on p. 9).

Chu, Yinghao et al. (Nov. 2018): "Multilayer Hybrid Deep-Learning Method for Waste Classification and Recycling". en. In: *Computational Intelligence and Neuroscience* 2018, pp. 1–9. ISSN: 1687-5265, 1687-5273. DOI: 10.1155/2018/5060857 (cit. on p. 6).

Coral-Support (Apr. 2020a): *Personal Communication*. Hello Michael -

Apologies for the issue and thanks for submitting the model. We've found and fixed the issue with our internal compiler. This will be fixed by the next compiler release; unfortunately we don't yet have a release data at this point.

Thanks, Coral Support Team!

Hello Michael,

There are no work arounds available for this issue for now. Please stay in touch with our website https://coral.ai/news/ for any updates.

Thanks Coral Support Team (cit. on p. 47).

Coral-Support (Apr. 2020b): *Personal Communication*. Hi Michael -

Thanks for reaching out!

The memory on the TPU is quite limited and I can't provide too much info into it. But the general idea is that not all of the input tensors can fit on the chip, so much of it will be held in the host's memory (you rpi). There are some changes we made between the std and the max frequencies that allows more data to be held on the TPU chips, which takes some load off from the host. This causes the TPU to run more frequently and produces more heat. Since libedgetpu is still not open source at this time, those are details that I can't provide, unfortunately.

Thanks, Coral Support Team! (Cit. on p. 61).

Costa, Bernardo S. et al. (Oct. 2018): "Artificial Intelligence in Automated Sorting in Trash Recycling". pt. In: *Anais do Encontro Nacional de Inteligência Artificial e Computacional (ENIAC)*, pp. 198–205. ISSN: 0000-0000. DOI: 10.5753/eniac.2018.4416 (cit. on p. 6).

Cyphers, Scott et al. (Jan. 2018): "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning". In: *arXiv:1801.08058 [cs]*. arXiv: 1801.08058 [cs] (cit. on p. 50).

Duboscq, Gilles et al. (Oct. 2013): "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, Indiana, USA: Association for Computing Machinery, pp. 1–10. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542143 (cit. on p. 50).

Garland, Michael et al. (July 2008): "Parallel Computing Experiences with CUDA". In: *IEEE Micro* 28.4, pp. 13–27. ISSN: 1937-4143. DOI: 10.1109/MM.2008.57 (cit. on pp. 10, 15).

Gyawali, Dipesh et al. (Apr. 2020): "Comparative Analysis of Multiple Deep CNN Models for Waste Classification". In: *arXiv:2004.02168 [cs]*. Comment: 6 pages, 13 figures. arXiv: 2004.02168 [cs] (cit. on p. 7).

Howard, Andrew G. et al. (Apr. 2017): "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". en. In: *arXiv:1704.04861 [cs]*. arXiv: 1704.04861 [cs] (cit. on pp. 22, 23).

Intel® Neural Compute Stick 2 Product Specifications (June 2020). en. https://ark.intel.com /content/www/us/en/ark/products/140109/intel-neural-compute-stick-2.html (cit. on p. 17).

Ionica, Mircea Horea and Gregg, David (Jan. 2015): "The Movidius Myriad Architecture's Potential for Scientific Computing". In: *IEEE Micro* 35.1, pp. 6–14. ISSN: 1937-4143. DOI: 10.1109/MM.2015.4 (cit. on p. 17).

Jacob, Benoit et al. (2018): "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713 (cit. on p. 16).

Jouppi, Norman et al. (May 2018): "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3, pp. 10–19. ISSN: 1937-4143. DOI: 10.1109/MM.2018.032271057 (cit. on pp. 10, 16).

Jouppi, Norman P. et al. (June 2017): "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12. DOI: 10.1145/3079856.3080246 (cit. on pp. 10, 16, 35).

Kaza, Silpa et al. (Dec. 2018): *What a Waste 2.0: A Global Snapshot of Solid Waste Management to 2050*. en. The World Bank. ISBN: 978-1-4648-1329-0 978-1-4648-1347-4. DOI: 10.1596/978-1-4648-1329-0 (cit. on p. 4).

Keras, Team (n.d.): *Keras Documentation: Keras Applications*. en. https://keras.io/api/applications/ (cit. on p. 21).

Kingma, Diederik P. and Ba, Jimmy (Jan. 2017): "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]*. Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. arXiv: 1412.6980 [cs] (cit. on p. 33).

Lee, Yen-Lin; Tsung, Pei-Kuei, and Wu, Max (Apr. 2018): "Techology Trend of Edge AI". In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–2. DOI: 10.1109/VLSI-DAT.2018.8373244 (cit. on p. 8).

Libutti, Leandro Ariel et al. (Jan. 2020): "Benchmarking Performance and Power of USB Accelerators for Inference with MLPerf". en. In: p. 15 (cit. on pp. 10, 16, 58, 61, 81, 83).

Model Optimizer Developer Guide - OpenVINO™ Toolkit (June 2020). https://docs.openvinotoolkit.org/latest/_docs_MO_DG_Deep_Learning_Model_Optimizer_DevGuide.html (cit. on p. 50).

Nickolls, John and Dally, William J. (Mar. 2010): "The GPU Computing Era". In: *IEEE Micro* 30.2, pp. 56–69. ISSN: 1937-4143. DOI: 10.1109/MM.2010.41 (cit. on p. 10).

Nvidia TensorRT Developer-Guide (June 2020). en-us. http://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html. Concept (cit. on p. 49).

Nvidia TF-TRT User-Guide (June 2020). en-us. http://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html. Concept (cit. on p. 49).

Ramachandran, Prajit; Zoph, Barret, and Le, Quoc V. (Oct. 2017): "Searching for Activation Functions". en. In: *arXiv:1710.05941 [cs]*. Comment: Updated version of "Swish: a Self-Gated Activation Function". arXiv: 1710.05941 [cs] (cit. on p. 31).

Rawat, Waseem and Wang, Zenghui (June 2017): "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review". In: *Neural Computation* 29.9, pp. 2352–2449. ISSN: 0899-7667. DOI: 10.1162/neco_a_00990 (cit. on p. 21).

Rivas-Gomez, Sergio et al. (May 2018): "Exploring the Vision Processing Unit as Co-Processor for Inference". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 589–598. DOI: 10.1109/IPDPSW.2018.00098 (cit. on pp. 10, 17).

Ross, Jonathan et al. (Nov. 2016): "Neural Network Processor". en. WO2016186801A1 (cit. on p. 16).

Sakr, George E. et al. (Nov. 2016): "Comparing Deep Learning and Support Vector Machines for Autonomous Waste Sorting". In: *2016 IEEE International Multidisciplinary Conference on Engineering Technology (IMCET)*, pp. 207–212. DOI: 10.1109/IMCET.2016.7777453 (cit. on p. 6).

Sandler, Mark et al. (June 2018): "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474 (cit. on pp. 22, 24, 28).

Shmueli, Boaz (June 2020): *Multi-Class Metrics Made Simple, Part II: The F1-Score*. en. https://towardsdatascience.com/ multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1 (cit. on p. 43).

Shung, Koo Ping (Apr. 2020): *Accuracy, Precision, Recall or F1?* en. https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9 (cit. on p. 43).

Sousa, João; Rebelo, Ana, and Cardoso, Jaime S. (Sept. 2019): "Automation of Waste Sorting with Deep Learning". In: *2019 XV Workshop de Visão Computacional (WVC)*, pp. 43–48. DOI: 10.1109/WVC.2019.8876924 (cit. on p. 6).

Tan, Mingxing; Chen, Bo, et al. (May 2019): "MnasNet: Platform-Aware Neural Architecture Search for Mobile". en. In: *arXiv:1807.11626 [cs]*. Comment: Published in CVPR 2019. arXiv: 1807.11626 [cs] (cit. on pp. 27, 28).

Tan, Mingxing and Le, Quoc V. (Nov. 2019): "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". en. In: *arXiv:1905.11946 [cs, stat]*. Comment: ICML 2019. arXiv: 1905.11946 [cs, stat] (cit. on p. 30).

Targ, Sasha; Almeida, Diogo, and Lyman, Kevin (Mar. 2016): "Resnet in Resnet: Generalizing Residual Architectures". In: *arXiv:1603.08029 [cs, stat]*. arXiv: 1603.08029 [cs, stat] (cit. on p. 22).

TensorFlow Lite Converter (June 2020). en. https://www.tensorflow.org/lite/convert (cit. on p. 45).

TensorFlow Models on the Edge TPU (June 2020). en-us. https://coral.ai/docs/edgetpu/mod els-intro/#compatibility-overview (cit. on p. 47).

Troubleshooting | Cloud TPU (Apr. 2020). en. https://cloud.google.com/tpu/docs/troubleshooting (cit. on p. 33).

Using the SavedModel Format | TensorFlow Core (June 2020). en. https://www.tensorflow.org/guide/saved_model (cit. on p. 45).

Vassanadumrongdee, Sujitra and Kittipongvises, Suthirat (Mar. 2018): "Factors Influencing Source Separation Intention and Willingness to Pay for Improving Waste Management in Bangkok, Thailand". en. In: *Sustainable Environment Research* 28.2, pp. 90–99. ISSN: 2468-2039. DOI: 10.1016/j.serj.2017.11.003 (cit. on p. 4).

Wang, Yu Emma; Wei, Gu-Yeon, and Brooks, David (Oct. 2019): "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning". en. In: *arXiv:1907.10701 [cs, stat]*. arXiv: 1907.10701 [cs, stat] (cit. on p. 35).

Zoph, Barret and Le, Quoc V. (Feb. 2017): "Neural Architecture Search with Reinforcement Learning". en. In: *arXiv:1611.01578 [cs]*. arXiv: 1611.01578 [cs] (cit. on p. 26).

Zoph, Barret; Vasudevan, Vijay, et al. (Apr. 2018): "Learning Transferable Architectures for Scalable Image Recognition". en. In: *arXiv:1707.07012 [cs, stat]*. arXiv: 1707.07012 [cs, stat] (cit. on pp. 26, 29).

Appendices

Code to create confusion matrix on Corals Edge TPU

```
""" Copyright 2019 The TensorFlow Authors. All Rights Reserved.

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

      https://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License."""

"Create Confusionmatrix from testset and measure CPU, Memory and Time consumption during inference"

"Import Packages"
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import os
import psutil
import time
import numpy as np
import cv2
import glob
import csv

"Import Tensorflow lite runtime interpreter"
for i in range(10):
    try:
        import tflite_runtime.interpreter as tflite
        break
    except:
       continue

"Define edge TPU libary"
EDGETPU_SHARED_LIB = 'libedgetpu.so.1'

"Function to load label names from textfile"
def load_labels(path):
    file = open(path)
    file = [line for line in file.readlines()]
    file = [s.rstrip() for s in file]
    return file

"Function to set the input tensor of the Tensorflow lite Edge TPU interpreter"
def set_input_tensor(interpreter, image):
  tensor_index = interpreter.get_input_details()[0]['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
  input_tensor[:, :] = image

"Function to create TensorFlow lite Edge TPU interpreter and initialize Coral TPU USB accelerator"
def make_interpreter(model_file):
  model_file, *device = model_file.split('@')
  return tflite.Interpreter(
      model_path=model_file,
      experimental_delegates=[
          tflite.load_delegate(EDGETPU_SHARED_LIB,
                               {'device': device[0]} if device else {})
      ])

"Function to classify image and return array of index of the class and confidence value"
def classify_image(interpreter, image, top_k=1):

  "Start interpreter for inference"
  set_input_tensor(interpreter, image)
  interpreter.invoke()
  output_details = interpreter.get_output_details()[0]
  output = np.squeeze(interpreter.get_tensor(output_details['index']))

  "If the model is quantized (uint8 data), then dequantize the results"
  if output_details['dtype'] == np.uint8:
    scale, zero_point = output_details['quantization']
```

```
76      output = scale * (output - zero_point)
77
78    ordered = np.argpartition(-output, top_k)
79    return [(i, output[i]) for i in ordered[:top_k]]
80
81
82  def main():
83
84      "Define path of the TensorFlow lite Edge TPU model"
85      model_path = 'model_edgetpu.tflite'
86
87      "Define path of the testset"
88      testpath = 'test/'
89
90      "Load label name form textfile"
91      labels_path = 'labels.txt'
92      labels = load_labels(labels_path)
93
94      "Calculate number of classes from labels"
95      numClass = len(labels)
96
97      "Create liste to collect time, cpu and memory data"
98      time_list = list()
99      cpu_list = list()
100     mem_list = list()
101
102     cpu_list.append(psutil.cpu_percent())
103     mem_list.append(psutil.virtual_memory()[2])
104
105     "Create and initialize interpreter of the Edge TPU"
106     interpreter = make_interpreter(model_path)
107     interpreter.allocate_tensors()
108     _, h, w, _ = interpreter.get_input_details()[0]['shape']
109
110     "Define image dimension"
111     dim =(h,w)
112
113     "Initialize run with class first index"
114     run =1
115
116     "Start loop througt classes"
117     while True:
118
119         "Clear list before collect data"
120         time_list.clear()
121         cpu_list.clear()
122         mem_list.clear()
123
124         "Initialize confusion matrix with zeros"
125         confMatrix = [[0 for i in range(numClass)] for j in range(numClass)]
126
127         "Initialize number of right and wrong prediction to zero"
128         right = 0
129         wrong = 0
130
131         for class_name in labels:
132         "Get path to folders with class images from testset"
133             test_dir = testpath + class_name
134             folders = os.listdir(test_dir)
135
136             "Loop througt folders with test images"
137             for folder in folders:
138                 imagePath = test_dir + '/' +folder
139                 images = [f for f in glob.glob(imagePath + "*/*.jpg", recursive=True)]
140
141                 "Loop througt images in folder"
142                 for imagePath in images:
143
144                     "Load image from path and resize to neuronal network input
                            dimension"
145                     img = cv2.imread(imagePath)
146                     img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
147
148                     "Start time measurement"
149                     start = time.time()
150
151                     "Set input data"
152                     input_data = np.expand_dims(img, axis=0).astype(np.uint8)
153
154                     "Start inference on Corals EdgeTPU"
155                     results = classify_image(interpreter, input_data)
156
157                     "Split result into index and confidence"
158                     top_k, prob = results[0]
159
```

```
                        "Stop time measurement"
                        stop = time.time()
                        time_list.append((stop-start)*1000)

                        "Calculate right and wrong predictions"
                        if labels[top_k] == class_name:
                            right = right +1
                        else:
                            wrong = wrong +1

                        "Add prediction to confusion matrix"
                        j = labels.index(class_name)
                        confMatrix[top_k][j] = confMatrix[top_k][j] +1

                        "Append cpu and memory to list"
                        cpu_list.append(psutil.cpu_percent())
                        mem_list.append(psutil.virtual_memory()[2])

            "Save confusionmatrix, cpu, memory and time list as CSV file"
            with open('ConfMatrix_' +str(run)+'.csv', 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerows(confMatrix)
                writer.writerow([right,wrong])

            with open('TimeList_' +str(run)+'.csv', 'w', newline='') as timefile:
                writer = csv.writer(timefile,quoting=csv.QUOTE_ALL)
                writer.writerow(time_list)

            with open('CPUList_' +str(run)+'.csv', 'w', newline='') as cpufile:
                writer = csv.writer(cpufile)
                writer.writerow(cpu_list)

            with open('mem_list_' +str(run)+'.csv', 'w', newline='') as memfile:
                writer = csv.writer(memfile)
                writer.writerow(mem_list)

            "Increment run to next class"
            run = run+1

            "Stop after last class"
            if run == numClass:
                break

if __name__ == '__main__':
    main()
```

Listing 1: Python code to create confusion matrix and collect time - cpu - memory datas on Corals Edge TPU

Code to create confusion matrix on Nvidias TensorRT Engine

```
""" Copyright 2019 The TensorFlow Authors. All Rights Reserved.

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

      https://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License."""

"Create Confusionmatrix from testset and measure CPU, Memory and Time consumption during inference"

"Import Packages"
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime
import os
import psutil
import time
import numpy as np
import cv2
import glob
import csv

"Import Tensorflow lite runtime interpreter"
for i in range(10):
    try:
        import tflite_runtime.interpreter as tflite
        break
    except:
       continue

"Define edge TPU libary"
EDGETPU_SHARED_LIB = 'libedgetpu.so.1'

"Function to load label names from textfile"
def load_labels(path):
    file = open(path)
    file = [line for line in file.readlines()]
    file = [s.rstrip() for s in file]
    return file

"Function to set the input tensor of the Tensorflow lite Edge TPU interpreter"
def set_input_tensor(interpreter, image):
  tensor_index = interpreter.get_input_details()[0]['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
  input_tensor[:, :] = image

"Function to create TensorFlow lite Edge TPU interpreter and initialize Coral TPU USB accelerator"
def make_interpreter(model_file):
  model_file, *device = model_file.split('@')
  return tflite.Interpreter(
      model_path=model_file,
      experimental_delegates=[
          tflite.load_delegate(EDGETPU_SHARED_LIB,
                               {'device': device[0]} if device else {})
      ])

"Function to classify image and return array of index of the class and confidence value"
def classify_image(interpreter, image, top_k=1):

  "Start interpreter for inference"
  set_input_tensor(interpreter, image)
  interpreter.invoke()
  output_details = interpreter.get_output_details()[0]
  output = np.squeeze(interpreter.get_tensor(output_details['index']))

  "If the model is quantized (uint8 data), then dequantize the results"
  if output_details['dtype'] == np.uint8:
    scale, zero_point = output_details['quantization']
```

```
76      output = scale * (output - zero_point)
77
78    ordered = np.argpartition(-output, top_k)
79    return [(i, output[i]) for i in ordered[:top_k]]
80
81
82  def main():
83
84      "Define path of the TensorFlow lite Edge TPU model"
85      model_path = 'model_edgetpu.tflite'
86
87      "Define path of the testset"
88      testpath = 'test/'
89
90      "Load label name form textfile"
91      labels_path = 'labels.txt'
92      labels = load_labels(labels_path)
93
94      "Calculate number of classes from labels"
95      numClass = len(labels)
96
97      "Create liste to collect time, cpu and memory data"
98      time_list = list()
99      cpu_list = list()
100     mem_list = list()
101
102     cpu_list.append(psutil.cpu_percent())
103     mem_list.append(psutil.virtual_memory()[2])
104
105     "Create and initialize interpreter of the Edge TPU"
106     interpreter = make_interpreter(model_path)
107     interpreter.allocate_tensors()
108     _, h, w, _ = interpreter.get_input_details()[0]['shape']
109
110     "Define image dimension"
111     dim =(h,w)
112
113     "Initialize run with class first index"
114     run =1
115
116     "Start loop througt classes"
117     while True:
118
119         "Clear list before collect data"
120         time_list.clear()
121         cpu_list.clear()
122         mem_list.clear()
123
124         "Initialize confusion matrix with zeros"
125         confMatrix = [[0 for i in range(numClass)] for j in range(numClass)]
126
127         "Initialize number of right and wrong prediction to zero"
128         right = 0
129         wrong = 0
130
131         for class_name in labels:
132         "Get path to folders with class images from testset"
133             test_dir = testpath + class_name
134             folders = os.listdir(test_dir)
135
136             "Loop througt folders with test images"
137             for folder in folders:
138                 imagePath = test_dir + '/' +folder
139                 images = [f for f in glob.glob(imagePath + "*/*.jpg", recursive=True)]
140
141                 "Loop througt images in folder"
142                 for imagePath in images:
143
144                     "Load image from path and resize to neuronal network input
                            dimension"
145                     img = cv2.imread(imagePath)
146                     img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
147
148                     "Start time measurement"
149                     start = time.time()
150
151                     "Set input data"
152                     input_data = np.expand_dims(img, axis=0).astype(np.uint8)
153
154                     "Start inference on Corals EdgeTPU"
155                     results = classify_image(interpreter, input_data)
156
157                     "Split result into index and confidence"
158                     top_k, prob = results[0]
159
```

```
                        "Stop time measurement"
                        stop = time.time()
                        time_list.append((stop-start)*1000)

                        "Calculate right and wrong predictions"
                        if labels[top_k] == class_name:
                            right = right +1
                        else:
                            wrong = wrong +1

                        "Add prediction to confusion matrix"
                        j = labels.index(class_name)
                        confMatrix[top_k][j] = confMatrix[top_k][j] +1

                        "Append cpu and memory to list"
                        cpu_list.append(psutil.cpu_percent())
                        mem_list.append(psutil.virtual_memory()[2])

            "Save confusionmatrix, cpu, memory and time list as CSV file"
            with open('ConfMatrix_' +str(run)+'.csv', 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerows(confMatrix)
                writer.writerow([right,wrong])

            with open('TimeList_' +str(run)+'.csv', 'w', newline='') as timefile:
                writer = csv.writer(timefile,quoting=csv.QUOTE_ALL)
                writer.writerow(time_list)

            with open('CPUList_' +str(run)+'.csv', 'w', newline='') as cpufile:
                writer = csv.writer(cpufile)
                writer.writerow(cpu_list)

            with open('mem_list_' +str(run)+'.csv', 'w', newline='') as memfile:
                writer = csv.writer(memfile)
                writer.writerow(mem_list)

            "Increment run to next class"
            run = run+1

            "Stop after last class"
            if run == numClass:
                break

if __name__ == '__main__':
    main()
```

Listing 2: Python code to create confusion matrix and collect time - cpu - memory datas on Nvidias TensorRT Engine

Code to create confusion matrix on Intels Openvino Engine

```
""" Copyright 2019 The TensorFlow Authors. All Rights Reserved.

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

      https://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License."""

"Create Confusionmatrix from testset and measure CPU, Memory and Time consumption
    during inference"

"Import Packages"
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from datetime import datetime

import os
import psutil
import time
import numpy as np
import cv2
import glob
import csv

"Import Openvino Inference Engine"
from openvino.inference_engine import IENetwork, IECore

"Function to load label names from textfile"
def load_labels(path):
    file = open(path)
    file = [line for line in file.readlines()]
    file = [s.rstrip() for s in file]
    return file


def main():

    "Files to create neuronal network architecture"
    model_xml = 'frozen_model.xml'
    model_bin = 'frozen_model.bin'

    "Define as VPU inference"
    device = 'MYRIAD'

    "Define path of the testset"
    testpath = 'test/'

    "Load label name form textfile"
    labels_path = 'labels.txt'
    labels = load_labels(labels_path)
    n_labels = len(labels)

    "Calculate number of classes from labels"
    numClass = len(labels)

    "Create liste to collect time, cpu and memory data"
    time_list = list()
    cpu_list = list()
    mem_list = list()

    cpu_list.append(psutil.cpu_percent())
    mem_list.append(psutil.virtual_memory()[2])

    "Initialize neuronal network architecture"
    ie = IECore()
    net = IENetwork(model=model_xml, weights=model_bin)

    assert len(net.inputs.keys()) == 1, "Sample supports only single input topologies"
    assert len(net.outputs) == 1, "Sample supports only single output topologies"

    input_blob = next(iter(net.inputs))
```

```
    out_blob = next(iter(net.outputs))
    net.batch_size = 1

    "Initialize Openvino Engine with Myriad VPU"
    n, c, h, w = net.inputs[input_blob].shape
    exec_net = ie.load_network(network=net, device_name=device)

    "Define image dimension"
    dim =(h,w)

    "Initialize run with class first index"
    run = 1

    "Start loop througt classes"
    while True:

        "Clear list before collect data"
        time_list.clear()
        cpu_list.clear()
        mem_list.clear()

        "Initialize confusion matrix with zeros"
        confMatrix = [[0 for i in range(numClass)] for j in range(numClass)]

        "Initialize number of right and wrong prediction to zero"
        right = 0
        wrong = 0

        for class_name in labels:
            "Get path to folders with class images from testset"
            test_dir = testpath + class_name
            folders = os.listdir(test_dir)

            "Loop througt folders with test images"
            for folder in folders:
                imagePath = test_dir + '/' +folder
                images = [f for f in glob.glob(imagePath + "*/*.jpg", recursive=True)]

                "Loop througt images in folder"
                for imagePath in images:

                    "Load image from path and resize to neuronal network input
                        dimension"
                    img = cv2.imread(imagePath)
                    img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

                    input_data = np.ndarray(shape=(n, c, h, w))
                    img = img.transpose((2, 0, 1))

                    "Start time measurement"
                    start = time.time()

                    "Set input data"
                    input_data = img

                    "Start inference on Openvino Engine"
                    results = exec_net.infer(inputs={input_blob: input_data})

                    "Get result from Inference"
                    results = results[out_blob]
                    result = results[0]
                    result = np.squeeze(result)
                    top_k = np.argsort(result)[-1]

                    "Stop time measurement"
                    stop = time.time()
                    time_list.append((stop-start)*1000)

                    "Calculate right and wrong predictions"
                    if labels[top_k] == class_name:
                        right = right +1
                    else:
                        wrong = wrong +1

                    "Add prediction to confusion matrix"
                    j = labels.index(class_name)
                    confMatrix[top_k][j] = confMatrix[top_k][j] +1

                    "Append cpu and memory to list"
                    cpu_list.append(psutil.cpu_percent())
                    mem_list.append(psutil.virtual_memory()[2])

        "Save confusionmatrix, cpu, memory and time list as CSV file"
        with open('ConfMatrix.csv', 'w', newline='') as file:
            writer = csv.writer(file)
```

```
162                 writer.writerows(confMatrix)
163                 writer.writerow([right,wrong])
164 
165             with open('TimeList_'+ str(run) +'.csv', 'w', newline='') as timefile:
166                 writer = csv.writer(timefile,quoting=csv.QUOTE_ALL)
167                 writer.writerow(time_list)
168 
169             with open('CPUList_'+ str(run) +'.csv', 'w', newline='') as cpufile:
170                 writer = csv.writer(cpufile)
171                 writer.writerow(cpu_list)
172 
173             with open('mem_list_'+ str(run) +'.csv', 'w', newline='') as memfile:
174                 writer = csv.writer(memfile)
175                 writer.writerow(mem_list)
176 
177             "Increment run to next class"
178             run = run+1
179 
180             "Stop after last class"
181             if run == numClass:
182                 break
183 
184 if __name__ == '__main__':
185   main()
```

Listing 3: Python code to create confusion matrix and collect time - cpu - memory datas on Intels Openvino Engine for Myriad VPU