Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering
2021/2022 Academic Year
March/April 2022 Graduation Session

Image Recognition Benchmark with Different Embedded Solutions: Google TPU, RockChip NPU, NVIDIA GPU

Supervisor	Candidate
Prof. Andrea CALIMERA	Davide Emanuele MICELI

Summary

Internet of Things has become more and more popular in recent years. Smart devices can send messages through the network, transfer a large variety of data, improve the quality of various services and security. With the rise of edge computing, many different IoT systems have been developed, and we now have the possibility to use them to work with deep neural networks. To make proper use of these devices, we need to know what are their limits and possibilities, and because each system has its unique characteristics, what can help is a series of experiments that make them in comparison under the same circumstances, to understand what are the advantages and disadvantages for each configuration. This work is an image recognition benchmark on different edge devices: Raspberry Pi, Coral USB Accelerator, Coral Dev Board, Rock Pi, and NVIDIA Jetson Nano. With the help of a test dataset and a selection of convolutional neural networks, we created a framework to compare the performances in terms of accuracy, inference time, and power consumption. From the experiments, it emerged that the Coral Dev Board is the fastest, Jetson Nano achieved the highest accuracy, and Rock Pi is the system that consumes less power during inference. Finally, we used what we learned to train a neural network with a self-made dataset for a future deployment in one of the systems we analyzed.

Acknowledgements

I would like to thank Giacomantonio Napoletano and Alessandro Stella, Deepware srl co-founders, who offered me an internship and proposed this thesis. Giacomantonio also followed me during the development of this work, suggesting how to proceed with the experiments.
I thank Josè Jaramillo, who also followed me from the beginning of this thesis. He helped me on all the technical difficulties and i learned a lot from him.
I thank Professor Andrea Calimera, my supervisor, who put his trust on this project.
I thank my parents, my grandmas, and my brother Alessandro, who helped me despite all difficulties, who always believed in me and gave their support in all my choices. Sharing this victory with you fills my heart with joy.
I thank my friends, with which i spent my spare time and gave me another reason to continue on this journey. In particular, I thank Francesco, for all the patience he had when we were roommates, and Enrico and Fadil, for the support they gave me in these five years and a half.
Finally, I would like to thank my beloved Germana, the person who knows the most about all my fears, my struggles, and the whole path that has brought me this far, but also the person that by staying by my side helped me overcome all the difficulties.
Thank you for all you have done. We finally did it.

Table of Contents

Chapter 1

Introduction

We live in the period of the spread of Industry 4.0. In recent years, concepts like Big Data, Artificial Intelligence, and the Internet of Things are becoming more common and have an impact on a lot of people's life. Think about all those devices of everyday life that from some time have become "smart". Thanks to technological progress, these objects are now equipped with a processing unit, which allows them to communicate with a network and carry out operations that some years ago were considered impossible. IoT means this, now even "things" can access the internet and this adds a completely new set of tools with very high potential. If we combine the world of IoT with that of AI, the result can be the application of deep learning algorithms, such as image recognition or object identification, voice recognition, in devices called Single-Board Computers (SBC), devices with very small size and therefore easily combinable with common objects. For example, smart cameras can identify the objects they are capturing and send a signal to the network about what they see. This kind of camera could be part of a safety system or it could be integrated into a car, which could be able to detect pedestrians thanks to object detection and avoid accidents (autonomous driving). It is therefore important to know more about IoT devices, the possibilities are countless not only in terms of security but also for other applications. Talking about Big Data, because the amount of data that circulates on the network is becoming higher and higher, if we let edge devices manage more operations we can create a

better network with improved capacity and generally more efficient [1, 2].
To give a contribution to the IoT field, in the past months we did a set of experiments to know more about this world and its current limits. In particular, our research focuses on the study of SBCs that are used to run deep learning algorithms.
In recent years we have seen the spread of numerous SBCs, developed by specialized companies and by brands well known to the public like Google. What many of the latest models have in common is the presence within them of a Neural Processing Unit (NPU), additional processors designed to run algorithms related to Machine Learning and Deep Learning. The computational power of NPUs, therefore, allows the boards to execute algorithms like image classification and object detection with high performances.
Given the variety of architectures on the market and their recent diffusion, we still don't have a clear idea of their specifications and the advantages that each board offers. It would be very useful to compare the different boards in terms of speed, power consumption, and accuracy of the neural networks, to understand what are the capabilities of the tools we have at the moment and what is most useful for our future projects. One solution could be to run tests on the boards of our interest, and if the tests are done for all the boards under the same conditions, so in the same environment and concerning the same variables, we would refer to a system of experiments called benchmark.
The idea of this thesis comes from the need to have more data about SBCs performances. Some of the boards already on the market have been selected (Raspberry Pi [3], CoralUSB Accelerator [4], Coral Dev Board [5], Rock Pi [6], NVIDIA Jetson nano [7]) and will be subject to several tests that will lead to the creation of a benchmark dataset. The task chosen for this research is image classification, when given an image and a neural network, the latter can classify the image as one of the classes of its knowledge. To do this benchmark, apart from the SBCs, two more elements are important, which can be changed, generating more sets of different experiments: the neural networks to be used and the dataset. Only one dataset was chosen for this research, which is

the famous ImageNet[8, 9], while three different neural networks were selected: ResNet [10], MobileNet [11], and EfficientNet [12]. The goal is therefore to collect enough data to be able to adequately compare the different devices and have an idea as precise as possible of the peculiarities of each board. For example, we could find that a certain board is quite fast, while another can make very accurate predictions or has very low power consumption. We hope the results will help make better future decisions for IoT-related projects and further speed up the development and the spreading of this new technology in the market. These tests will be followed by an application on a real case, with a completely different dataset created for the occasion and using an appropriate combination of board and neural network, decided based on the results obtained. All the experiments were performed using the Python programming language, some of its most famous libraries such as Keras [13], TensorFlow [14], NumPy [15] but also other libraries created specifically to be able to comfortably use the boards in the Artificial Intelligence context.

Chapter 2

Background

Before proceeding with the description of the experiments it is appropriate to make an explanation of all the arguments that will be treated. This chapter will be used to describe the knowledge necessary for the development of the experiments, the tools we have used, and the research material to which we have easy access. Being able to combine all these elements is the result of the past two years of study, in fact all the topics we are going to describe were treated during the master's degree course. Also the computer engineering background I have proved to be useful, as it made learning these concepts easier thanks to previous knowledge of mathematics, statistics, and object-oriented programming.

2.1 Machine Learning and Deep Learning

2.1.1 What is machine learning

Let's start by talking about the branch of computer science which is the reason why we are interested in this benchmark, that is the possibility of implementing artificial intelligence algorithms in edge devices. To explain what machine learning is, it is appropriate to explain how it differs from classic programming. To do that, François Chollet in his book "Deep Learning with Python" [16] compares the classic paradigm of programming with the machine learning one. He says that in the old paradigm, humans create systems that, receiving an input and a set of

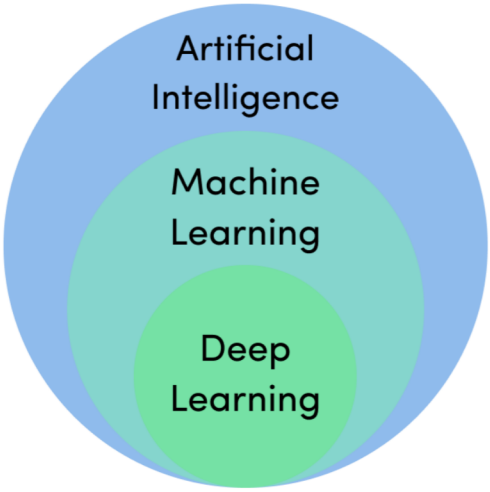

Figure 1: programming diagram from [17]

rules, also defined by humans, have the objective to generate a result. In Machine Learning, on the other hand, a system receives input and the expected result, so its objective is to define the rules that relate the input to the result. Machine learning is an evolution of the old paradigm: a machine learning algorithm is used to create an intelligent system that learns the rules autonomously, without humans having to do it.
To make this happen, a machine learning algorithm must be trained, that is, it must become aware of the type of input we are interested in and, for each input, the result we expect. For example, if you want to create an algorithm that can distinguish spam e-mails from good ones (ham), you have to give the system several e-mails (input) with the attached type, spam or ham (result), so that by analyzing the data can understand what are the characteristics of the two classes of mail. When the training is over we will have a system that, receiving an email as input, will be able to tell us if it is a spam email or a ham email. The operation of giving a result based on input and the rules learned by the system is called *inference*, one of the core concepts for the next chapters.

Another example of the application of machine learning is the weather forecast, where the algorithm receives temperature measurements taken periodically and is able to predict what the next measurement will be.

2.1.2 Artificial Neural Networks

Machine Learning models use advanced statistical methods to define the rules we need. However, as the complexity of the task increases, these models turn out to be inadequate, therefore it is necessary to resort to other solutions, represented in this case by neural networks.

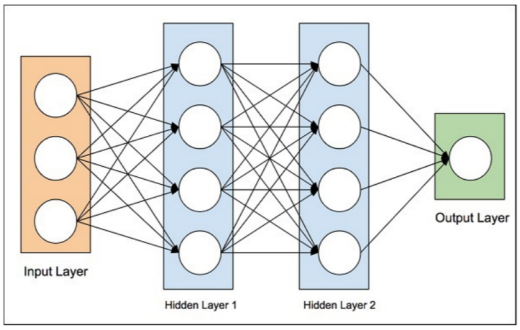

Figure 2: Example of Artificial Neural Network from [18]

The name "neural" takes inspiration from the neurons of the human brain, which by receiving stimuli can generate others. Similarly, a neural network is made up of neurons that communicate with each other creating a path through which input is transformed into an output. Neurons, or nodes, are organized in groups called layers. Each layer processes the data it receives and sends the result to the next layer through the links between the nodes.
So we have an input layer, where the input is divided into different portions. the nodes of the next layer receive some input portions and assign a weight to each of them, after which if the total received value exceeds a certain threshold, the node transfers the information to the next layer, otherwise, it does not transfer anything. The data then goes through all these filtering steps that define the importance of each portion of the data, until it reaches the output layer where a result is decreed. During training, the network receives numerous inputs that are used to calibrate the weights assigned by the nodes and to define the connections between the layers, so that each input can be converted

into the correct output. The system just described is called an artificial neural network, composed of an input layer output layer and 1 or 2 internal layers called hidden layers. These are hidden because we only know the initial data and the result, while what happens inside the network is unknown to us: we do not know what is the state of data in the central part, because it is the result of operations that the network defines autonomously during training, which is why we usually refer to the inner part of a neural network as "black box".

2.1.3 Deep Convolutional Neural Networks for Image Classification

As we said, artificial neural networks are very simple networks, made by only a few number of layers. A more complex network, with a high number of layers, is instead called a Deep Neural Network. These are systems that are used for tasks where the input consists of a large number of values, which cannot be simplified by a simple artificial neural network.
One of the tasks for which it is necessary to use a deep neural network is image classification. It consists in assigning the correct category or class to which each input image belongs. Each image of the training set is therefore associated with a class, and the network must learn the rules that are used to recognize each class. For example, if we have a training set with two different classes, the network output will consist of two nodes, one corresponding to the first class and one to the second. Only one of those nodes activates, determining the class of the image; this is an example of binary classification.
For what concerns deep neural networks in the context of image classification, we can divide the neural network into two parts: the feature extractor and the classifier. The feature extractor is the component whose job is to extract crucial information from an input, which might determine the class. We have already said that in an artificial neural network, at each piece of data is given a weight. Following the same reasoning, because in an image there are portions that are more important than others, the task of the feature extractor is to identify the

most important characteristics of the input, and then pass them to the classifier. The classifier, on the other hand, is the component that, looking at the characteristics identified by the feature extractor, can understand at which class they belong to and declare it in the last layer of the neural network.
In the experiments carried out for this research, we used deep neural networks which are also "convolutional", we will now explain what this last term means. The term convolutional is used to refer to those neural networks that have convolutional layers, which are used to extract features from an image. Convolutional layers apply filters to image's pixels. These filters are used to search for spatial properties between neighboring elements, because they may suggest the existence of an important feature in that area. Convolution is an operation that is applied to the majority of numerical values of the image and consists of a series of algebraic operations concerning the current pixel and a variable number of pixels around it. The result of each convolution is a single value that depends on the properties of the central pixel and its neighbors. In figure 3 we can see an example of a 3x3 convolution, done in the pixel in second row and second column.

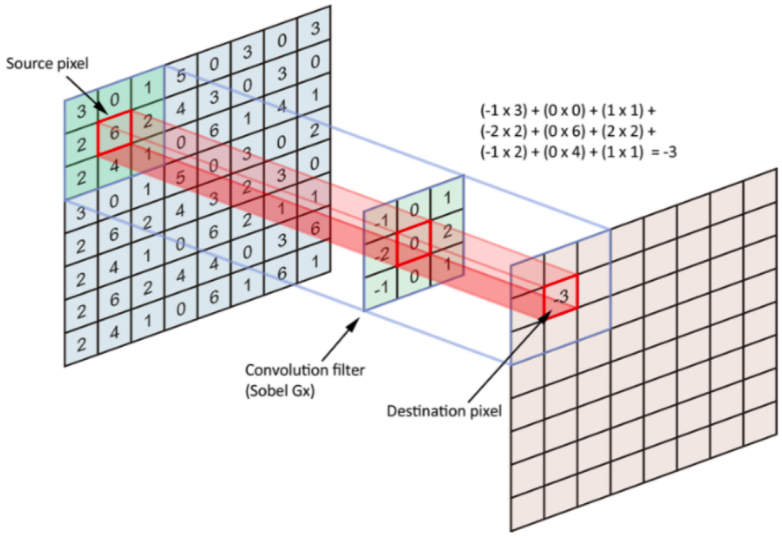

Figure 3: Convolution example from [19]

Each selected cell is multiplied by the corresponding kernel value, after which all values are added together to give a single final value. The kernel values are defined during training, where the network understands what filters it needs to find the features of each class. In a neural network for image classification, each convolution layer usually reduces the size of the image but highlights its main properties.
After the classifier has analyzed the features, it assigns to each class a probability that the image belongs to it, so after the classification, we will have a vector of n elements, where n is the number of classes of the task, formed by values ranging from 0 to 1 representing the probability. The predicted class is the one that has the highest value in this vector. We can see in [20] one of the first approaches to image classification with the use of neural networks.

2.2 Neural Networks

We will now proceed to a brief description of the three convolutional neural networks that we used in our experiments. The first network was the ResNet50 [10], while later we focused on more recent networks and close to the state of the art, that is Mobilenet, both the first [11] and the second version [21], and three different versions of EfficientNet [12].

2.2.1 ResNet50

The ResNet is a convolutional neural network presented at the 2016 CVPR. It is available in different versions which differ by number of layers and number of nodes. The size of a neural network must be chosen based on the complexity of the task, therefore different versions of the ResNet have been designed so that this network can be used in as many situations as possible. Some examples are Resnet18, ResNet32, ResNet50, ResNet 101, but knowing the structure of the layers that make up the ResNet it is also possible to create a customized version from scratch with the preferred number of layers. To give a general idea of how to choose the width of a network, we can say that if in

an image classification task where we have very small images as input, such as 32x32, we can start with the experiments from a ResNet18 or ResNet32 rather than a ResNet101. However, this is only a general guideline that may not apply in all cases.
The version chosen for this research is the one with 50 layers, one of the most used and with great support from the research community. The feature extractor is composed of standard 3x3 convolutions, followed by the classifier which consists of a single classification layer. The name of this network is an abbreviation of Residual Network because the concept of residuals is fundamental.

For a brief explanation, following the nomenclature of ResNet papers, we call x the input data of a given layer, this data will be transformed by one or more layers by a function F (), which will lead to having the result F(x), to which the input data x, the residue, can be added. Storing the data x in the form of a residue can help in the optimization of the network, because in this way it is possible, if the circumstances require it, to skip all the operations of a given layer by setting its weights to 0 and continue in the network using the residue as the next input.

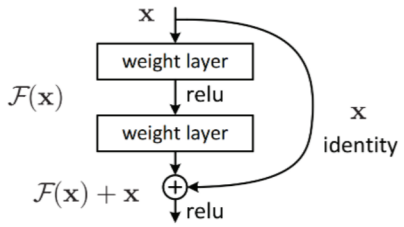

Figure 4: ResNet section from [10]

After the first tests carried out on this network, it was considered necessary to have other options, as will be shown in the results in the following paragraphs.

2.2.2 MobileNet

MobileNet was the second network selected for this project. It is a convolutional network much lighter than ResNet50, both for the lower number of layers and an effective technique that made it possible to reduce the size. In each layer, the standard convolution has been replaced with the combination of a depthwise convolution and the 1x1 or pointwise convolution. As the researchers who worked on this project say, A standard convolution borh filters and combines inputs into a new set of outputs in one step. The depthwise separable convolution splits this into two layers, a layer for filtering and one for combining. This method leads to a significant lowering of computational costs while maintaining high-level accuracy in the model.

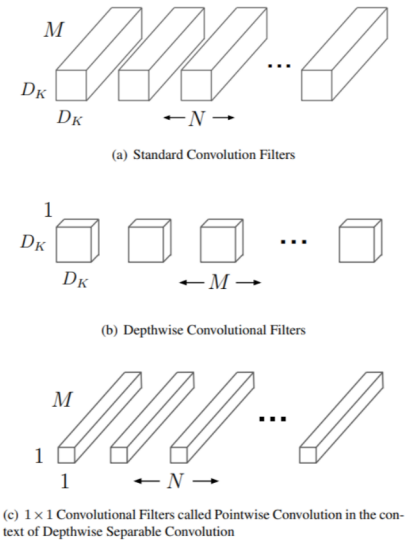

Figure 5: MobileNet convolutions from [11]

2.2.3 EfficientNet

The last network analyzed is also the most recent. EfficientNet is a family of convolutional neural networks of different sizes, all available on Keras (from EfficientNetB0 to EfficientNetB7). The need to create so many versions of the same network comes from the fact that, as we have already said, different tasks require networks of different sizes, so the creators, in their research, focused on finding the most efficient way to modify the size of a network and make it a tool as versatile as possible.
Starting from the basic model shown in the EfficientNet paper [12], to create subsequent versions it is necessary to increase the depth, width,

and resolution of the input according to a certain criterion, which in this case is the same for all variants. After several experiments, the creators have found that the most efficient method to increment the three values just mentioned is to raise them to a common value ϕ, as shown in figure 6.

$$\begin{aligned} \text{depth: } & d = \alpha^{\phi} \\ \text{width: } & w = \beta^{\phi} \\ \text{resolution: } & r = \gamma^{\phi} \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

Figure 6: EfficientNet equations from [12]

In this research, we will find results for EfficientNetB0 (S), B1 (M), and B3 (L), as we have found to be the most popular versions. In particular, EfficientNetB0 has a 224x224 input dimension, the same as ResNet and MobileNet, which is great for comparing those networks with each other.

2.3 Dataset

Another important choice for our image classification benchmark is the dataset, which can be divided into training set, test set, and validation set. In our context a dataset is a collection of data used to train Machine Learning and Deep Learning algorithms, we will now describe all the possible usages for image classification. The first possible use of a dataset is to provide data for training, so that the network will use it to learn to recognize the different classes, this data is part of the training set. We remember that the goal of a neural network is, for each class, to identify features that are common to all the images of that class, so that if we provide as input an image that was not used during training, the network is still able to identify the class, otherwise, we

face the phenomenon called overfitting, which means that the network works well only in with images used in training, i.e. it is not able to generalize enough to recognize also other images. To determine if the network has succeeded in the task of avoiding overfitting, we use a new set of data that was not present during the training, called the test set. The test set is given as input to the trained network, to see how accurate it is with data never seen before. The ideal situation occurs when the same accuracy results are achieved in both training and test set, but this is too difficult and in many situations, it is normal to accept lower accuracy results on the test set.
Sometimes a third type of dataset is used, to test the network in the middle of training to avoid overfitting, this dataset is called validation set. During training, the network takes into account the accuracy of the test set, which is one of the factors that influence how the parameters change during training.

2.3.1 Imagenet

Regarding our experiments, we need a difficult task to understand what are the limits not only in terms of board speed but also in terms of accuracy. In this regard, we have chosen the famous ImageNet [9] as the dataset for all our tests. Imagenet is a dataset created to provide the research community with a strong reference point for experiments of various nature.
It was created following the WordNet hierarchy [22] Which contains over 100,000 synsets, each describing a meaningful concept in English. The goal for Imagenet was to provide a good number of images from each synset, and as a result, Imagenet is a database made of millions of images. For what concerns the image classification task, 1000 classes, not hierarchically dependent on each other have been selected; this is our reference dataset.
Although the original ImageNet was not used directly in this research, all the neural networks of this project were pre-trained using the image classification version of ImageNet.

2.3.2 ImageNetv2

For the test dataset, the one used for our benchmark, we chose another version of Imagenet, more recent and with a smaller size.
It is called Imagenetv2 [23], another set of images belonging to the same 1000 classes of ImageNet, with 10 images per class. Imagenetv2 was created after a data collection process that began a decade after Imagenet's publication, which makes the two datasets distinct from each other, so there is no risk of overfitting when using ImageNetv2 as a test set. After collecting all the images, 10 candidates per class were selected, resulting in a total of 10,000 images. To choose the 10 candidates for each class, creators used three different criteria, resulting in the creation of three different versions for ImageNetv2 [24].

2.4 Python

As already mentioned, this is a computer science project, so we need a programming language to carry out all the experiments. For this purpose, we have chosen Python.
Python is an object-oriented programming language, widely used in Machine Learning and Deep Learning. Since it is a high-level programming language, we have numerous libraries with pre-built classes and functions that provide excellent tools to work easily in this field. Among the most used libraries for this research we find:

- Numpy[15], which provides numerous mathematical functions and methods of managing Python data structures;
- Keras [13] and Tensor Flow [14], which are libraries made for the implementation of deep learning algorithms. They offer a series of methods to manage the entire deep learning pipeline, from the definition of a neural network to the setup for training and test of neural networks;
- Pycoral [25], rknn-toolkit [26], and TensorRT [27] are the libraries specifically designed to work with the boards we have selected.

From them, we used all the functions necessary for the inference tests.

2.5 GitHub

Now we can talk about a resource that turned up to be crucial for this project. As for the code part, we must say that it was not written all from scratch. On Web, we have the opportunity to find a huge amount of work done by other programmers, which is available as open-source, so for the base structure of a script, you can choose to start from someone else's work and then add your contribution. After all, it is not necessary to reinvent the wheel, especially if it would lead to a delay on the schedule and a waste of time that we could have used to do more experiments. It is for this reason that most of the scripts that we will see are made using as a starting point some works we found on the Github platform [28]. GitHub is a place where numerous projects are collected in the form of repositories. It is an interface that makes it easier to access jobs that, for the versioning aspect, are based on git, the most efficient system for coordinating different users who are working on the same project. Inside GitHub, it is possible to find files made by private users, code related to published papers, and official repositories of various companies, which are published to provide examples and tutorials for users who approach a particular topic for the first time. This last category is what we are interested in. Each company that has released a board that we have analyzed has a repository where we can find several introductory examples, useful to familiarize ourselves with the devices and their libraries.
We want to give credit to the original sources of the script we will see later, so they are all grouped here [25] [26] [29] [30] [31] [32], although before each script the source will be again specified.

2.6 What we tested

To conclude this chapter, we will show the systems we have analyzed. For each of them, there will be a brief description of their characteristics, followed by some technical specifications that allow us to make a first estimate of the results we are going to obtain.

2.6.1 Coral USB Accelerator and Coral Dev Board

Google Coral is a family of devices for AI acceleration. Among these, we have selected the USB accelerator and the Dev Board. The former is a device made only by a Tensor Processing Unit (TPU), so its only purpose is to do tensor operations. For this reason, to use it, it must be connected to a host device that redirects all the calculations to the accelerator. The Dev Board is instead an SBC, so it can be used autonomously. this is the substantial difference we find between the two Coral devices. From the benchmark perspective, the two devices are more or less the same, in fact we used the same benchmark script for both. However, we will notice some differences in the results.

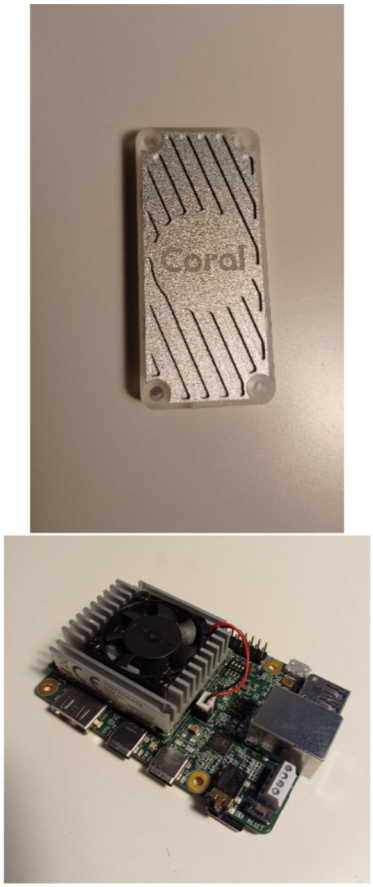

Figure 7: Google Coral USB Accelerator and Dev Board

As the technical page says, Both the USB accelerator and the Dev Board have 4 Tensor Operations Per Second (TOPS) and 2

TOPS per watt, while the Application Programming Interface (API) used on python is pycoral, which requires the use of neural networks in tflite format. Pycoral accounts for all the operations required for inference: from loading the model to printing the results.

2.6.2 Raspberry Pi 4 Model B

To carry out the experiments on Coral USB Accelerator we have chosen Raspberry Pi 4, which allows the installation of pycoral. In particular, the model we used is the one with 4 GB of RAM. We decided to run the experiments also on the Raspberry alone, which is an exception in the context of this benchmark. The reasons are that it is a very popular device, pretty familiar to the public, and to have the possibility to highlight the difference in terms of performances between it and NPU devices. The Raspberry Pi is used for many applications in the IoT world, a lot of customization options are possible through different sensors sold separately, which allow its use in many situations. Among the most common we cite the temperature/humidity sensor, microphone, and the PiCamera. As we will see later, the experiments are carried out using the same framework as the Coral devices.

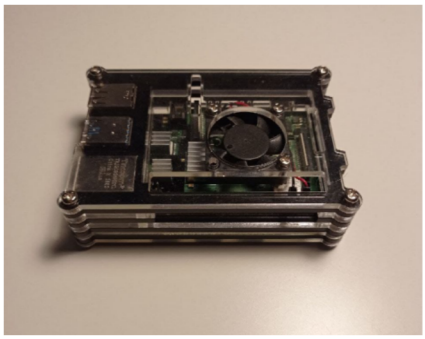

Figure 8: My Raspberry Pi 4

2.6.3 Rock Pi N10 A

Rock Pi N10 is one of the SBCs with integrated NPU released by Radxa. The model available to us is version A, the smallest among the three available, with 4 GB of RAM, including 1 GB used for NPU operations. The processor is the RK3399pro, one of the best released by this manufacturer. The board supports different types of model formats (pb, tflite, onnx, caffe) and the API used is rknn-toolkit, which has all the necessary methods for inference.

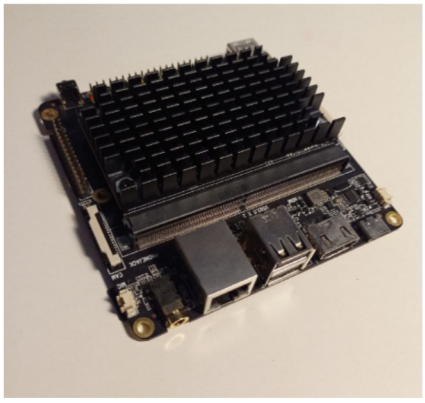

Figure 9: Rock Pi N 10

2.6.4 NVIDIA Jetson Nano

One of NVIDIA's proposals for Deep Learning on embedded systems is Jetson Nano, the latest board of our research. This company has been offering for several years various solutions for deep learning through the use of GPUs, such as the GeForce series for Personal Computers. This allowed the creation of a common framework for deep learning called NVIDIA Jetpack SDK, also available for our board, together with the possibility of using CUDA kernels, another common system among NVIDIA devices.

Figure 10: NVIDIA Jetson Nano

2.6.5 Summary of technical specifications

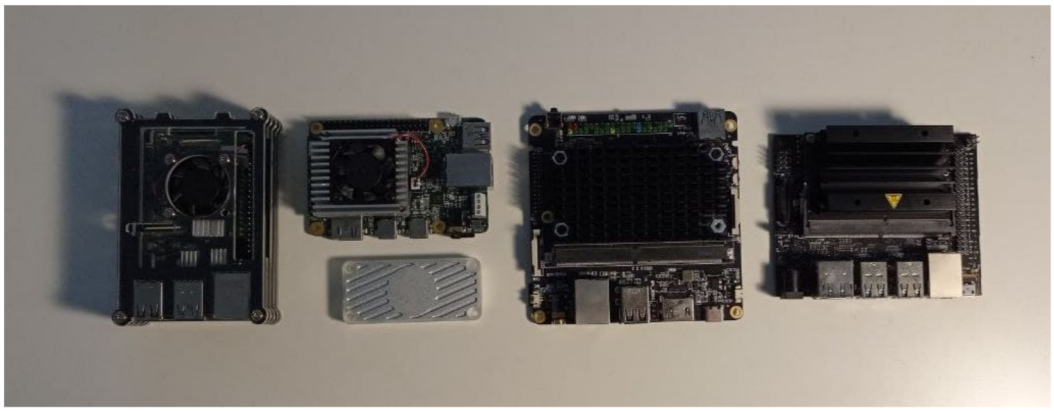

Figure 11: A comparison of the size of the devices. From left to right: Raspberry Pi, Coral Dev Board and USB Accelerator, Rock Pi, NVIDIA Jetson Nano

Let's now take a look at the technical specifications, with which it is possible to make the first comments.

	Raspberry Pi	Coral USB Accelerator	Coral Dev Board	Rock Pi N10 A	NVIDIA Jetson Nano
Hardware	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)	ML Accelerator: Google Edge TPU coprocessor: **4 TOPS** (int8); 2 TOPS per watt	CPU: NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F) GPU: Integrated GC7000 Lite Graphics ML Accelerator: Google Edge TPU coprocessor: **4 TOPS** (int8); 2 TOPS per watt	CPU: Dual Cortex-A72, frequency 1.8Ghz with quad Cortex-A53, frequency 1.4Ghz GPU: Mali T860MP4, OpenGL ES 1.1 /2.0 /3.0 /3.1 /3.2, Vulkan 1.0, Open CL 1.1 1.2, DX11 NPU: Support 8bit/16bit computing, up to **3.0TOPs** computing power	GPU: 128-core NVIDIA Maxwell™ architecture-based GPU CPU: Quad-core ARM® A57 NPU: **472 GFLOPS**
API	pycoral	pycoral	pycoral	rknn-toolkit	jetson-inference, TensorRT
model precision	int8	int8	int8	int8	**fp32**
model format	.tflite	edge-tpu.tflite	edge-tpu.tflite	.pb, .onnx, .caffe, .tflite	.onnx, .trt

Looking at the technical datasheet, we see that Jetson Nano has the slowest NPU (472 GFLOPS), but it is the only device we can use to work with floating-point numbers, for which we expect excellent results in terms of accuracy.
We also expect good results in terms of speed from both Coral devices, and we will see if Rock Pi flexibility on model format will play an important role.

Chapter 3

Related Works

The world of System on Chip (SoC) and IoT is generally very complex and full of challenges [33]. It is very important to explore this field, as the development of edge computing, complementary to that of cloud computing, facilitates the diffusion of many innovative applications [1, 2]. Many researchers have dedicated their time studying these tools and analyzing the potential of both IoT systems and individual Boards. This chapter is used to describe researches that are similar to ours. The contribution of the researchers who published the following papers allowed us to understand many of the limitations of IoT networks and individual boards, some of which were also selected for our project. First, we will see works that define a benchmark system for IoT devices, then we will continue with the works more focused on the performance of different SBCs, which are the most related.

IoT devices, apart from Deep Learning, can be used for many different applications, many of which involve the transmission of a large amount of data. RIoTBench, described in [34] is a study on the use of Distributed Stream Processing Systems (DSPS) for streaming IoT applications. This benchmark covers 27 different IoT tasks divided into 4 different categories. For each task, latency, throughput, jitter, and memory utilization were analyzed, while for the implementation, they used Virtual Machines with Intel Xeon processors.

IoTABench [35] Is an IoT benchmark toolkit that can be adapted to

different use cases, based on the user's needs. In the paper, its usage is demonstrated in the application of smart metering, done after the generation of a large amount of synthetic data. The experiments took place in a cluster of 8 HP servers.

An important element of our research is the application of deep learning models. This is important not only in SBCs but also in other systems. [36] is a study about the application of deep learning models in Leading-edge systems.

We now describe works that have an objective similar to ours. They are not the first approach of deep learning in edge devices, like in [37], but they follow the same line and add measurements about inference time and power consumption. In [38] We see a Benchmark on SBCs ASUS Tinker, Raspberry Pi 4, Coral Dev board, NVIDIA Jetson Nano, Arduino Nano 33 BLE. They analyze the power consumption during the experiments, while the MobileNet network is used to calculate inference time and accuracy. The images for the tests are taken from the Imagenet validation set. In particular, 5000 of the images of this set are used.

The study in [39] focuses on Coral Dev Board, USB Accelerator, Jetson Nano, and Intel NCS. They stored results for memory usage, execution time, and energy consumption.

Finally, in [40] we can see a comparison between the Coral dev board and Jetson Nano, for which they calculated memory and power consumption, inference time, and accuracy. The accuracy is calculated on 100 images using Inception and GoogLeNet as neural networks. These two networks were also used in inference time tests together with different versions of MobileNet.

At this point, we can ask ourselves what is the goal of our work compared to those just presented. The aim is to collect data that are useful to better understand the boards and improve their usage. In this regard, we created a framework that helps us understand the behavior of different combinations of board models datasets and neural networks, so that each test gives a result for each of these configurations. We

analyze a model already seen in other works, which is Mobilenet, but we introduce tests on ResNet50 and EfficientNet. Accuracy is calculated on Imagenetv2 and a new board is introduced, which is Rock Pi N10. For these last two elements, we are not aware of studies where they have been applied before.

Chapter 4

Implementation

In this chapter we will first talk about the Key Performance Indicators, that is the characteristics of the boards for which we are interested in collecting data. Then we will show the implementation of the experiments, where each paragraph is dedicated to a different board and divided into an initial part about setup, followed by the description of the code and the actual tests. Finally, we will talk about the training carried out for our dataset of rings using the transfer learning technique. Many times during this chapter, the term x86 will be used to refer to the personal computer with Windows OS that was used for some preliminary operations before the actual tests, and subsequently also for the final training. All the boards are Linux-based systems, including Debian and Ubuntu builds.

4.1 Key Performance Indicators

As we have already said, our benchmark is concerned with measuring several parameters simultaneously. We decided to focus our research on measuring accuracy, inference time, and power consumption.

- **Accuracy**: it is the percentage of images in the test set that the neural network can classify correctly. In our case, it is therefore a ratio between the number of correct predictions and the 10.000 ImageNetv2 images. Although it may seem that the same neural network gives the same results regardless of which board it is deployed on, this is not always true, as the same network in different formats can give different results. Measuring the accuracy will help to understand if the limitations of our SBCs can affect the accuracy of some neural networks.

- **Inference time**: for each image of the test set we measure the time the neural network takes to give a result. We will then have 10,000 measurements for each experiment, for which we will show the average value.

- **Power Consumption**: There is a clear difference between the consumption of a board in an idle state and the consumption while running an application. With this metric, we want to identify the amount of this increase, trying to understand what are the factors that determine it. By doing that, we can also calculate the total consumption during the entire inference cycle. This will help to understand what are the limits in terms of autonomy.

4.2 Raspberry Pi and Coral

Since the Raspberry Pi and Coral Devices are used in parallel for most of the experiments involving them, for a better understanding, this section will describe the work of all three of them.

4.2.1 Raspberry Pi setup for Coral USB Accelerator

Before using the USB accelerator it is necessary to perform some setup operations in the Raspberry. The first thing to do is to install Edge TPU Runtime, adding the corresponding repository to the system, and updating with sudo apt-get update. Then we can install the libedgetpu1-std library and connect the accelerator to the Raspberry. Finally, we install the pycoral API with apt-get.

4.2.2 Coral Dev Board Setup

First, we need to download the image file containing the device's operating system. It is possible to use BalenaEtcher from a PC to install the file inside a microSD. Meanwhile, the board, which has not be alimented yet, must be set in microSD boot mode, correcting the position of the dedicated four switches if necessary. When the microSD is ready, it can be inserted into the board and the latter powered. In this phase the board is installing the files contained in the microSD into the eMMC memory, so that, after this operation, the eMMC boot mode can be set, returning the switches in their original position. For convenience reasons, we preferred to work with both the accelerator and the Dev Board via Raspberry, to manage the two devices from a single host and to be able to perform some actions in parallel. To do this it is necessary to install Mendel-development-tool (mdt) on the Raspberry, a command-line tool that allows interacting with the Dev Board. The two devices are put into communication via USB/micro-USB cable. After which we can use the mdt start command to access the Dev Board

terminal from the Raspberry. [1]

4.2.3 Models

Pycoral is an API created from Tensor Flow Lite. For this reason, there is a lot in common between the methods of the two libraries. For example, to use pycoral the models has to be in tflite format. There is a difference between the models used to test the Raspberry and those used for Coral devices: we use standard tflite models for the Raspberry, while for Coral we have to compile the models into edgeTPU format, which is required by Coral TPUs. The MobileNets and EfficientNets that we decided to use are available on GitHub [43], Both in tflite and edge-tpu.tflite format. For the ResNet50, it was necessary to compile the model manually, which can be done in two ways: it is possible to install the edgetpu-compiler on the device of interest if it matches the requirements, or we do the conversion on Google Colab with a specific script [32]. From x86 we used a script to load the model directly from Keras and convert it into tflite, while the second conversion in edgetpu was made using Google Colab. Another very important requirement of pycoral is quantization: all models must be converted into int8 precision in order to be used. MobileNet and EfficientNet models were ready to use, while for ResNet50 the quantization was done during the tflite conversion.

4.2.4 About quantization

For a correct understanding of a factor that is important both in this and in the following sections, it is convenient to do a small digression to clarify the concept of quantization.
In the IoT world, memory and speed optimization are often important. In our Deep Learning context, this means that we have to make sure that neural networks are as efficient as possible. One of the techniques that save memory and make networks generally faster is quantization.

[1] for more information about setup, please visit [41] and [42]

Quantization refers to the optimization process that involves the shift of neural network elements from one type of precision format to another. For example, it is possible to quantize a network from fp32 precision to int8, which generally makes it more suitable for deployment on IoT devices. As we will see later, quantization plays a very important role in many of the tests performed. We have already seen that Coral devices require int8 quantization, we will see what changes where we do not have this constraint.

4.2.5 Code

The creation of the code starts from a script available in the official GitHub repository of pycoral [25], where we can find examples about image classification, object detection, semantic segmentation, and many other tasks. After installing the repository requirements with *install-requirements.sh*, we tried the script *classify-image.py*. It receives a network and an image to be classified as arguments, then the inference is done five times and we can see the time for each try. The chosen class and the respective confidence are printed at the end. From this point, we need to modify the code to let ImageNetv2 be the input, calculate the inference times and total accuracy. To have an average of the single inference times, the body of the code is put inside many cycles as the number of images in the test set. To measure the inference time, we store the execution time of the *invoke()* method, which is used to perform the prediction. Each inference time does not take into account the time needed to load the input and its preprocessing but only the moment of prediction, as shown in the following code excerpt.

```
start = time.perf_counter()
interpreter.invoke()
inference_time += time.perf_counter()-start
```

It is important to note that different models may have different output ranges. For example, the ResNet output goes from 0 to 999, where the first class is identified with 0 and the last with 999, while the range of the MobileNet goes from 1 to 1000. Since the script is unique and

receives the model path as an argument, it is necessary to check the output range of the model we are going to use inside the code. It is possible to obtain this information using *getoutputdetails()* and add the appropriate offset to the result of the prediction since the ground truth range is always (0.999). For example, if we are using a Mobilenet and we are making a prediction for a class 0 image, a correct result of the MobileNet would be 1, so you must always add 1 to the value of the output to see if there is a match with the result.

The same code can be used for all three configurations described in this paragraph: Raspberry Pi, Coral USB, and Coral Dev Board. In the case of the Raspberry Pi + Coral tests, if the model received from the command line is a standard tflite, the inference is done on the Raspberry, while if the model is an edge-tpu.flite all the operations are automatically redirected to the USB accelerator. With the Dev Board, we only test edgetpu models, which are treated in the same way as in USB Accelerator. This automation is possible thanks to the pycoral framework, built to manage the calculations sending them to the correct unit.

```
# Lint as: python3
# Copyright 2019 Google LLC
#
# 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.


import argparse
import time
import glob
import numpy as np
from PIL import Image
from pycoral.adapters import classify
from pycoral.adapters import common
from pycoral.utils.dataset import read_label_file
from pycoral.utils.edgetpu import make_interpreter
import sys


def main():

  NUM_CLASSES = 1000
  IMG_PER_CLASS = 10
  offset = 0
  top_k = 1
  threshold = 0.0
```

```
36  inference_time = 0
37  predicted = 0
38
39  parser = argparse.ArgumentParser(
40      formatter_class=argparse.ArgumentDefaultsHelpFormatter)
41  parser.add_argument(
42      '-m', '--model', required = True, help = 'File path of
     .tflite file.')
43  parser.add_argument(
44      '-i', '--input', required = True, help = 'Dataset to be
      classified.')
45
46  args = parser.parse_args()
47
48  interpreter = make_interpreter(*args.model.split('@'))
49  interpreter.allocate_tensors()
50
51  #different models have different output range, the
     following variable is used to check if the range is
     (0:999) or (1:1000)
52  output_range = interpreter.get_output_details()[0]['
     shape_signature'][1]
53  if output_range > NUM_CLASSES:
54      offset = 1
55  # Model must be uint8 quantized
56  if common.input_details(interpreter, 'dtype') != np.uint8:
57    raise ValueError('Only support uint8 input type.')
58
59  size = common.input_size(interpreter)
60
61  for cl in range(1000):
62
63      class_images = glob.glob(f"{args.input}/{cl}/*")
64
65      for n in range(10):
66
67          image = Image.open(class_images[n]).convert('RGB').
     resize(size, Image.ANTIALIAS)
68          common.set_input(interpreter, image)
69
70          # Run inference
71          start = time.perf_counter()
```

```
72            interpreter.invoke()
73            inference_time += time.perf_counter() - start
74            classes = classify.get_classes(interpreter, top_k,
    threshold)
75            if classes[0].id == cl + offset:
76                predicted += 1
77
78        sys.stdout.write(f"\rdoing inference: {cl+1}/1000
    classes done")
79
80    print('\n\n----------RESULTS----------')
81    print(f'AVG INFERENCE TIME:{inference_time /10} ms')
82    print(f'ACCURACY: {predicted/NUM_CLASSES*IMG_PER_CLASS}')
83
84 if __name__ == '__main__':
85    main()
```

4.3 Rock Pi N10 A

4.3.1 Setup

All setup operations are described on the official Radxa forum [44], Where we can also find links to the required .whl files. After downloading the image file and preparing the microSD, we need to update the operating system, including the Radxa repository. This is followed by the installation of the NPU management unit and the initialization of the NPU, required at each boot-up. We then proceed with the installation of several python libraries, i.e. all TensorFlow and rknn-toolkit requirements, before proceeding with the installation of these two libraries. The version we choose for rknn-toolkit is 1.4.0. It is not the latest, but it is the most stable in terms of dependencies with other libraries. The TensorFlow version is 1.14.0 also for dependency reasons. In fact, after trying to use a tensorflow2 build, which would have facilitated some of the subsequent operations, we encountered numerous conflicts that led to a fresh reinstallation of the operating system, after which all the recommended versions were installed correctly.

4.3.2 Models and code

For the MobileNets, we used the tflite version previously tested on Coral, while for the ResNet50 we have to load it in .pb format and then do the conversion in .rknn, which is required by rknn-toolkit.

By the time of writing, there is not a method to test the EfficientNets on Rock Pi.

We can find a repository containing some introductory examples also for this board [26]. In order to perform inference with rknn-toolkit these are the steps to follow:

- creation of the RKNN() object, which will manage all the operations related to inference;
- configuration and preprocessing with rknn.config(): allows to set the values for normalization, rearrange the input channels, and choose the type of quantization;
- loading the model: a different method is used for each format. For tflite models we only need to specify the file path as argument, while for models in .pb format we must also define the input name, the output name, and input size;

```
ret = rknn.loadtensorflow (
    tfpb = "ResNet50.pb",
    inputs = ["input1"],
    outputs = ["fc1000/Softmax"],
    inputsizelist = [[224,224,3]]
)
```

- building of the model with rknn.build (), where the model is also quantized if necessary;
- exporting the model in the required format (rknn) with rknn.exportrknn()

- preprocessing of the input image with other libraries, in this case, cv2;
- starting the runtime with rknn.initruntime ()
- inference with rknn.inference ()

The only differences between the code for ResNet and the MobileNet one are the model loading function, the output range control, and the quantization, which is needed only if we use ResNet, as MobileNets are already quantized.
The quantization of ResNet50 is a process that takes several minutes but is required only once, as the model is subsequently saved in .rknn format so ready to be used for other code executions. Initially, the quantization was never successful: after a few minutes from the beginning of the quantization, the execution always got interrupted. After a brief analysis of the code, it seemed like there weren't any issues related, so the next attempt was to monitor the RAM consumption while the script was running. On Linux systems, we can use the top command, which allows this type of control. In this way, we found that the RAM consumption increases dramatically at the time of quantization and that 4 GBs are not enough. So the process gets interrupted when the RAM reaches its maximum capacity. The solution to this problem was to use a swap file, which is a portion of memory used to store data that cannot stay in the RAM because of its saturation. In this way, we finally can finish the quantization. The swap file was created using part of the microSD memory. It was initially 2GB large, but when we found that the process requires around 6GB of RAM in total, we opted for a 4GB swap file. Since this type of memory is significantly slower than RAM's, we have noticed a further slowdown in the process, but as mentioned above, we do this operation one per model. Although it was not necessary, I also tried to quantize MobileNet to see if we encounter the same problem. Given the small size, I expected a lower RAM requirement, as well as a shorter execution time. The quantization of such a small model does not require the use of a swap file. We do not have a precise threshold that indicates when it is necessary to use it, but in general, we could say that if we plan to use a large network we

should be prepared to address the RAM saturation issue.
After testing ResNet and MobileNets, the last group of networks remaining is the EfficientNets, but unfortunately, the experiments on them have not been completed. Here we can see all the changes made for EfficientNet, starting with the ones regarding the model structure:

- As for MobileNet, for which we didn't find any issues, we tried to use the same tflite model used for the raspberry;
- We tested a tflite EfficientNet made by converting a Keras model from x86;
- We tried to download the tflite version of EfficientNet directly from TensorflowHub, both the int8 version and the fp32 to be quantized;
- We tried to get the .pb version of the model with the same procedure used for ResNet, but there was a problem with the output name, which has to be specified in *rknn.loadtensorflow()*. During the conversion to .pb it is possible to see the name of the network output, but when the same name is declared in *loadtensorflow()*, the program throws an error because it cannot find a layer with that name;

All the tests described above were carried out in conjunction with the following changes to the code

- Since one of the first operations that EfficientNet does is to normalize the input, We tried to remove normalization from *rknn.config()*;
- We tried to change the channel order of the input image, both from *rknn.config()* and with a *cv2* method called during the input definition phase;
- For the preprocessing before the inference we first used cv2, then Pillow.Image() and finally the same preprocessing that is done for EfficientNet on Jetson Nano
- The model was built with and without quantization.

We are currently waiting for a response from the Radxa Forum [45], The only portal we have access to, where we have explained the problem.

```
1
2  # Copyright 2020 Rockchip Electronics Co.,Ltd.
3  # All rights reserved.
4  #
5  # Redistribution and use in source and binary forms, with or
       without
6  # modification, are permitted provided that the following
       conditions are met:
7  #
8  # 1. Redistributions of source code must retain the above
       copyright notice,
9  # this list of conditions and the following disclaimer.
10 #
11 # 2. Redistributions in binary form must reproduce the above
       copyright notice,
12 # this list of conditions and the following disclaimer in the
        documentation
13 # and/or other materials provided with the distribution.
14 #
15 # 3. Neither the name of the copyright holder nor the names
       of its contributors
16 # may be used to endorse or promote products derived from
       this software without
17 # specific prior written permission.
18 #
19 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
20 # AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
21 # IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE
22 # ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
       CONTRIBUTORS BE
23 # LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
       EXEMPLARY, OR
24 # CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
       PROCUREMENT OF
```

```
25 # SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS
       ; OR BUSINESS
26 # INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY
       , WHETHER IN
27 # CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
      OR OTHERWISE)
28 # ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
       ADVISED OF THE
29 # POSSIBILITY OF SUCH DAMAGE.
30
31 import time
32 import platform
33 import numpy as np
34 import cv2
35 from rknn.api import RKNN
36 import glob
37 import argparse
38 import sys
39
40
41 if __name__ == "__main__":
42     parser = argparse.ArgumentParser(formatter_class =
      argparse.ArgumentDefaultsHelpFormatter)
43     parser.add_argument(
44             '-m', '--model', required=True, help= 'File path
      of .tflite file.')
45     parser.add_argument(
46             '-i', '--input', default="./imagenetv2_top", help
      = 'Path of dataset directory.')
47     parser.add_argument("--quant", action='store_true', help=
      "whether of not to quantize the model with rknn framework,
       only ResNet needs to be quantized")
48     parser.add_argument("--mobilenet",action="store_true")
49     args = parser.parse_args()
50     offset = 0
51     if args.mobilenet:
52         offset = 1
53     # Create RKNN object
54     rknn = RKNN()
55     img_height = 224
56     rknn.config(
57             quantized_dtype="asymmetric_quantized-u8",
```

```
    )


    # Direct Load RKNN Model
    ret = rknn.load_tensorflow(
            tf_pb="ResNet50.pb",
            inputs=["input_1"],
            outputs=["fc1000/Softmax"],
            input_size_list=[[224,224,3]]
            )
    if ret != 0:
        print(f"Load {args.model} failed!")
        exit(ret)
    if args.quant:
        ret = rknn.build(do_quantization=True, dataset = "./
dataset-samples.txt")
    else:
        ret = rknn.build(do_quantization=False)
    if ret!=0:
        print(f"Build {args.model} failed!")
        exit(ret)


    # init runtime environment
    print("--> Init runtime environment")

    if "aarch64" in platform.platform():
        target = "rk3399pro"
    else:
        target = None

    ret = rknn.init_runtime()
    if ret != 0:
        print("Init runtime environment failed")
        exit(ret)
    # Set inputs
    predicted = 0
    times = []
    for cl in range(1000):
        files = glob.glob(f"{args.input}/{cl}/*")
        for n in range(10):
            img = cv2.imread(files[n])
```

```
99              img = cv2.resize(img, dsize=(img_height,
    img_height))
100
101
102             img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
103
104
105
106             # Inference
107             start_time = time.time()
108             outputs = rknn.inference(inputs=[img])
109             delta = time.time() - start_time
110             times.append(delta)
111
112             pred = np.array(outputs).argmax()
113             print(pred,cl)
114             if pred == cl + offset:
115                 predicted += 1
116
117         sys.stdout.write(f"\rdoing inference: {cl+1}/1000
    classes done")
118
119
120     # Calculate the average time for inference.
121     mean_delta = np.array(times).mean()
122     fps = 1 / mean_delta
123
124     print("\n\n--------RESULTS--------")
125     print("average(sec):{:.3f},fps:{:.2f}".format(mean_delta,
    fps))
126     print(f"accuracy: {predicted/100}%")
127     rknn.release()
```

4.4 NVIDIA Jetson Nano

4.4.1 Setup

There are two ways to give power to the Jetson Nano. You can choose whether to use the appropriate jack port or the micro-USB input. If you choose the first option and keep the second free, it can be used to communicate with a host using software like PuTTy [46]. After installing the operating system via microSD we proceed with the initial configuration, where we accept the terms of service and we set the language, time, default keyboard, and administrator account. After these steps, we will already have JetPack installed [47], a package containing NVIDIA's API, TensorRT. Another important element is CUDA [48], installed separately: it is used to manage the tensor with the use of GPU. Then we proceed with the download of the repositories that we will use as a starting point for the inference.

4.4.2 Models and Code

The scripts are a modified version of examples taken from two different repositories. The first one [29] is installed with *cmake*, which offers some customization regarding additional libraries and models to download. Among the prebuilt models, we also find ResNet50. To test the ResNet50 we need to modify the example script to receive ImageNetv2 as input and calculate the KPIs.
This script works with functions from the jetson-inference class, a high-level wrapper of the original TensorRT API, which is written entirely in C ++.
For the experiments on MobileNets and EfficientNets, we start from another repository [30], which uses the actual Python API. Here is a brief description of the pipeline that can be implemented with TensorRT: we start with the creation of the logger, the unit used to print errors above a certain severity. We proceed with the initialization of the builder, which is used to load and build the onnx model. Models in onnx format must be converted to engine format before being used on tensorRT, so

this is the next step. This is followed by the runtime creation, input definition, and inference. The test script for EfficientNet and MobileNet skips the conversion step from onnx to engine, so we need to have a model in engine format before running the test. To do this I used two other scripts from the same repository: the first performs the conversion from pb. to onnx, while the second from onnx to engine. The pb model can be obtained by saving a Keras model in saved model format. This will generate a folder with a name defined by the user, where inside we will find the file we are interested in called savedmodel.pb, which is used as input for the first conversion. the entire conversion process takes from 30 minutes for MobileNet to approximately 90 minutes for EfficientNetB3. Between the two conversions, the one that converts a .pb model to onnx is the slowest. The models are never quantized, so we find them with fp32 precision in engine format.

```
1  #
2  # Copyright (c) 2021, NVIDIA CORPORATION. All rights reserved
       .
3  #
4  # Licensed under the Apache License, Version 2.0 (the "
       License");
5  # you may not use this file except in compliance with the
       License.
6  # You may obtain a copy of the License at
7  #
8  #     http://www.apache.org/licenses/LICENSE-2.0
9  #
10 # Unless required by applicable law or agreed to in writing,
       software
11 # distributed under the License is distributed on an "AS IS"
       BASIS,
12 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
       express or implied.
13 # See the License for the specific language governing
       permissions and
14 # limitations under the License.
15 #
16
17 import os
```

```
18 import sys
19 import argparse
20
21 import numpy as np
22 import tensorrt as trt
23
24 import pycuda.driver as cuda
25 import pycuda.autoinit
26
27 from image_batcher import ImageBatcher
28 import glob
29 import time
30
31 class TensorRTInfer:
32     """
33     Implements inference for the EfficientNet TensorRT engine.
34     """
35
36     def __init__(self, engine_path):
37         """
38         :param engine_path: The path to the serialized engine to load from disk.
39         """
40         # Load TRT engine
41         self.logger = trt.Logger(trt.Logger.ERROR)
42         with open(engine_path, "rb") as f, trt.Runtime(self.logger) as runtime:
43             self.engine = runtime.deserialize_cuda_engine(f.read())
44         self.context = self.engine.create_execution_context()
45         assert self.engine
46         assert self.context
47
48         # Setup I/O bindings
49         self.inputs = []
50         self.outputs = []
51         self.allocations = []
52         for i in range(self.engine.num_bindings):
53             is_input = False
54             if self.engine.binding_is_input(i):
55                 is_input = True
```

```
56              name = self.engine.get_binding_name(i)
57              dtype = self.engine.get_binding_dtype(i)
58              shape = self.engine.get_binding_shape(i)
59              if is_input:
60                  self.batch_size = shape[0]
61              size = np.dtype(trt.nptype(dtype)).itemsize
62              for s in shape:
63                  size *= s
64              allocation = cuda.mem_alloc(size)
65              binding = {
66                  'index': i,
67                  'name': name,
68                  'dtype': np.dtype(trt.nptype(dtype)),
69                  'shape': list(shape),
70                  'allocation': allocation,
71              }
72              self.allocations.append(allocation)
73              if self.engine.binding_is_input(i):
74                  self.inputs.append(binding)
75              else:
76                  self.outputs.append(binding)
77
78          assert self.batch_size > 0
79          assert len(self.inputs) > 0
80          assert len(self.outputs) > 0
81          assert len(self.allocations) > 0
82
83      def input_spec(self):
84          """
85          Get the specs for the input tensor of the network.
    Useful to prepare memory allocations.
86          :return: Two items, the shape of the input tensor and
     its (numpy) datatype.
87          """
88          return self.inputs[0]['shape'], self.inputs[0]['dtype
    ']
89
90      def output_spec(self):
91          """
92          Get the specs for the output tensor of the network.
    Useful to prepare memory allocations.
```

```
93          :return: Two items, the shape of the output tensor and its (numpy) datatype.
94          """
95          return self.outputs[0]['shape'], self.outputs[0]['dtype']
96
97      def infer(self, batch, top=1):
98          """
99          Execute inference on a batch of images. The images should already be batched and preprocessed, as prepared by
100         the ImageBatcher class. Memory copying to and from the GPU device will be performed here.
101         :param batch: A numpy array holding the image batch.
102         :param top: The number of classes to return as top_predicitons, in descending order by their score. By default,
103         setting to one will return the same as the maximum score class. Useful for Top-5 accuracy metrics in validation.
104         :return: Three items, as numpy arrays for each batch image: The maximum score class, the corresponding maximum
105         score, and a list of the top N classes and scores.
106         """
107         # Prepare the output data
108         output = np.zeros(*self.output_spec())
109
110         # Process I/O and execute the network
111         cuda.memcpy_htod(self.inputs[0]['allocation'], np.ascontiguousarray(batch))
112         start = time.time()
113         self.context.execute_v2(self.allocations)
114         delta = time.time()-start
115         cuda.memcpy_dtoh(output, self.outputs[0]['allocation'])
116
117         # Process the results
118         classes = np.argmax(output, axis=1)
119         scores = np.max(output, axis=1)
120         top = min(top, output.shape[1])
121         top_classes = np.flip(np.argsort(output, axis=1), axis=1)[:, 0:top]
```

```
122             top_scores = np.flip(np.sort(output, axis=1), axis=1)
    [:, 0:top]
123
124             return classes, scores, [top_classes, top_scores],
    delta
125
126
127 def main(args):
128     NUM_CLASSES=1000
129     IMG_PER_CLASS=10
130     predicted = 0
131     total_time = 0
132     offset = 0
133     if args.mobilenet:
134         offset = 1    #mobilenets have output range (1,1000)
135     trt_infer = TensorRTInfer(args.engine)
136     for cl in range(NUM_CLASSES):
137         batcher = ImageBatcher(args.input + f"/{cl}", *
    trt_infer.input_spec(), preprocessor=args.preprocessor)
138         for batch, images in batcher.get_batch():
139             classes, scores, top ,delta= trt_infer.infer(
    batch)
140             total_time += delta
141             for i in range(len(images)):
142                 if classes[i] == cl + offset: #adjust the
    range of ground truth if the network is mobilenet
143                     predicted += 1
144                 if args.top == 1:
145                     print(images[i], classes[i], scores[i],
    sep=args.separator)
146                 else:
147                     line = [images[i]]
148                     assert args.top <= top[0].shape[1]
149                     for t in range(args.top):
150                         line.append(str(top[0][i][t]))
151                     for t in range(args.top):
152                         line.append(str(top[1][i][t]))
153                     print(args.separator.join(line))
154     print(f"accuracy: {predicted/NUM_CLASSES*IMG_PER_CLASS}")
155     print(f"inference time: {total_time/10} ms")
156 if __name__ == "__main__":
157     parser = argparse.ArgumentParser()
```

```
158     parser.add_argument("-e", "--engine", help="The TensorRT
    engine to infer with")
159     parser.add_argument("-i", "--input",
160                         help="The input to infer, either a
    single image path, or a directory of images")
161     parser.add_argument("-t", "--top", default=1, type=int,
162                         help="The amount of top classes and
    scores to output per image, default: 1")
163     parser.add_argument("-s", "--separator", default="\t",
164                         help="Separator to use between
    columns when printing the results, default: \\t")
165     parser.add_argument("-p", "--preprocessor", default="V2",
        choices=["V1", "V1MS", "V2"],
166                         help="Select the image preprocessor
    to use, either 'V2', 'V1' or 'V1MS', default: V2")
167     parser.add_argument("--mobilenet",action="store_true")
168     args = parser.parse_args()
169     if not all([args.engine, args.input]):
170         parser.print_help()
171         print("\nThese arguments are required: --engine and
    --input")
172         sys.exit(1)
173     main(args)
```

4.5 Power consumption

This section will describe the methodology we used to measure power consumption, which is the same for all devices. Including the power consumption measurements is important to understand how much energy we need for the task we are interested in. We will see the change in consumption from idle to inference mode, as well as if there is a relationship between consumption and neural network size.

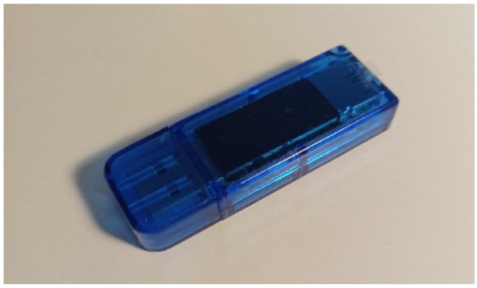

Figure 12: USB Tester

First, let's see how the devices are powered. Since we are dealing with modern SBCs, we only need one mobile phone power adapter. In particular, the one we used has a USB port, in which we can connect a USB/type-C or USB/micro-USB cable. With these tools, we can power up all the boards of our research. At this point, we need a device specifically designed to measure the power generated by this power supply: the consumption analysis was possible thanks to an additional instrument capable of detecting voltage, current, and consequently the power consumed, the USB Tester by RuiDeng.

It has a USB input and output so that it can be inserted between the power supply and the USB cable. Looking at the device display (figure 13) we can see the values we are interested in, like the voltage and current on the left, and in the lower part, from left to right, the total power consumed and the current average power consumption. Since it is not possible to save all the readings of the device in a file, the measurement of the power can be considered an estimate, since it is made by looking at the values in the display of the tester.

Figure 13: USB Tester display

However, it can be said that the

following methodology is suitable
to get a clear idea of the power consumed by the devices: it is possible to reset the counter of the total power consumed at the desired moment. In our case, the counter is set to zero at the start of the inference script, so that when it ends, the value we see on the counter will be the total power consumed during the execution of the script, then we divide it by the total execution time to get the average power consumption. Although this method is subject to some human error, for our research we can accept the uncertainty of a few mW that come with this measurement method. The results we will see on consumption will be in Watts, with a precision of one decimal place, so that we can guarantee the reliability of the data we will see. For the measurements in idle state, we deal with values that are much less subject to fluctuations, so a simple reading of the current power from the tester is sufficient. With a decimal digit precision, our measurements for the idle state are in line with what we see in [39].
For each board, we can measure the consumption directly from the link to the power adapter. The only exception is for the Coral USB accelerator, connected to the raspberry which is then connected to the power supply. Since there is no situation in which it is possible to use the accelerator alone, we take as a reference the power from the raspberry side, as shown in figure 14, although we measured also the Accelerator consumption.

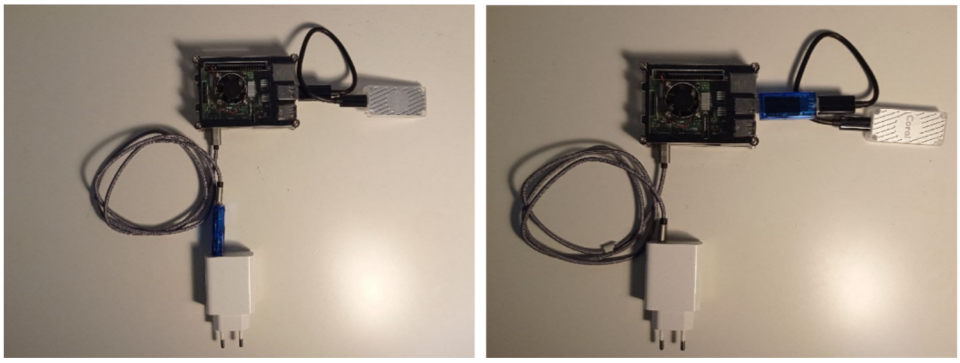

Figure 14: tester on raspberry side and on Coral side

4.6 Training neural networks for rings dataset

The last set of experiments is about an application on a real case of some of the tools we explored during previous tests. The objective is to deploy a deep learning model in an edge device for ring recognition. For this purpose, we created a dataset of 100 classes one for each ring. Each class has 32 images, one for different orientations and positions of the ring, which were captured by two cameras that made the captures in a controlled environment. In particular, the photos were all taken at the same brightness and with the same background, to recreate the environment in which we expect the model will be applied. The aim of the following experiments is to understand the complexity of the task using different neural networks, to finally choose the most suitable board for the deployment.

4.6.1 About Transfer Learning

We now proceed by explaining a technique we used in this phase which has facilitated the learning of the rings classes by neural networks.
Reminding some concepts explained in the background chapter, we know that a neural network for image classification can be divided into feature extractor and classifier, with the purpose of feature extractor being to identify particular characteristics of the classes we analyze. By doing that the feature extractor learns to recognize different shapes and constructs, and because some of these features can be useful in some other situations, experts have wondered if this kind of knowledge can be transferred into another training for a new dataset [49]. This is how transfer learning was born. It consists of training on a new dataset using part or all of a feature extractor that has been previously trained on another dataset, and what we practically do is to take a pre-trained network, freeze a portion of layers and then train only the final ones. When a layer is frozen its weights do not change in value during training, we do it to maintain the knowledge that the network has acquired by training with the old dataset.

We tried to do transfer learning starting from the neural networks we used for the benchmark, in their pre-trained version with ImageNet, keeping all the layers of the feature extractor frozen and only training the classifier. Transfer learning is particularly effective when the dataset we want to train with has a small amount of data and it is in some way similar to the one neural network was initially trained on. These requirements fit particularly well with our situation.

4.6.2 Training description

We now give a brief description of the training script, run on x86.
The following training was possible thanks to the work made by Yixing Fu in 2020 that we can find in [31].
the first few lines deal with loading the dataset. In particular, the image_dataset_from_directory () method is used. For a correct use of this method, the dataset was organized into a series of folders, each containing a different class. We can see here the first part of the script, where we can also have a look at some training images.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
from tensorflow import keras

 # IMG_SIZE is determined by EfficientNet model choice
IMG_SIZE = 224
batch_size = 32
NUM_CLASSES = 104

import pathlib
data_dir = "./Images"
data_dir = pathlib.Path(data_dir)
image_count = len(list(data_dir.glob('*/*.jpg')))

train_ds = tf.keras.preprocessing.
    image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
```

```
  subset="training",
  seed=42,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=batch_size)

val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=42,
  image_size=(IMG_SIZE, IMG_SIZE),
  batch_size=batch_size)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")
```

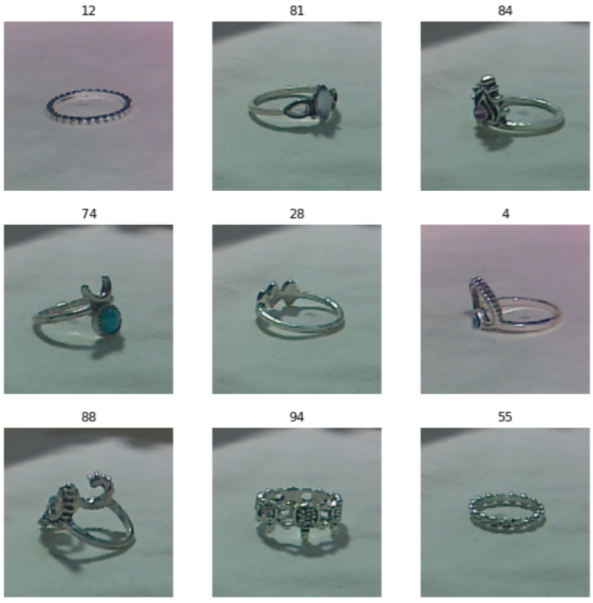

Figure 15: Some images from the rings dataset

The following part is where the network is loaded, in this example we see EfficientNetB0. Note that the classifier is not included (include_top = False), we will recreate it from scratch, because we need to train the classifier and because we have a different number of classes, 104 against 1000 classes for ImageNet.

After we freeze all the feature extractor layers, the new classifier is created and linked to the rest of the network with tf.keras.Model (). We then proceed with the compilation of the model and with the actual training using fit().

```
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.applications.efficientnet import (
    preprocess_input,
    decode_predictions,
)
from tensorflow.keras import layers
def build_model(num_classes):
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
    x = preprocess_input(inputs)
```

```
    model = EfficientNetB0(include_top=False, input_tensor=x,
    weights="imagenet")

    # Freeze the pretrained weights
    model.trainable = False

    # Rebuild top
    x = layers.GlobalAveragePooling2D(name="avg_pool")(model.
   output)
    x = layers.BatchNormalization()(x)

    top_dropout_rate = 0.2
    x = layers.Dropout(top_dropout_rate, name="top_dropout")(
   x)
    outputs = layers.Dense(NUM_CLASSES, activation="softmax",
    name="pred")(x)

    # Compile
    model = tf.keras.Model(inputs, outputs, name="
   EfficientNet")
    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-2)
    model.compile(
        optimizer=optimizer, loss="
   sparse_categorical_crossentropy", metrics=["accuracy"]
    )
    return model

import matplotlib.pyplot as plt


def plot_hist(hist):
    plt.plot(hist.history["accuracy"])
    plt.plot(hist.history["val_accuracy"])
    plt.title("model accuracy")
    plt.ylabel("accuracy")
    plt.xlabel("epoch")
    plt.legend(["train", "validation"], loc="upper left")
    plt.show()

model = build_model(num_classes=NUM_CLASSES)

epochs = 25
```

```
hist = model.fit(train_ds, epochs=epochs, validation_data=
    val_ds, verbose=2)
plot_hist(hist)
```

```
Epoch 1/25
63/63 - 109s - loss: 3.7723 - accuracy: 0.3741 - val_loss:
    2.2509 - val_accuracy: 0.4008
Epoch 2/25
63/63 - 109s - loss: 1.1580 - accuracy: 0.7286 - val_loss:
    1.4438 - val_accuracy: 0.5992
Epoch 3/25
63/63 - 110s - loss: 0.7674 - accuracy: 0.8207 - val_loss:
    1.4389 - val_accuracy: 0.6052
Epoch 4/25
63/63 - 108s - loss: 0.5993 - accuracy: 0.8503 - val_loss:
    1.3991 - val_accuracy: 0.6232
Epoch 5/25
63/63 - 109s - loss: 0.5186 - accuracy: 0.8708 - val_loss:
    1.4150 - val_accuracy: 0.6633
Epoch 6/25
63/63 - 108s - loss: 0.5347 - accuracy: 0.8743 - val_loss:
    1.4454 - val_accuracy: 0.7034
Epoch 7/25
63/63 - 109s - loss: 0.4763 - accuracy: 0.8878 - val_loss:
    1.6066 - val_accuracy: 0.7054
Epoch 8/25
63/63 - 118s - loss: 0.4595 - accuracy: 0.8933 - val_loss:
    1.3635 - val_accuracy: 0.7395
Epoch 9/25
63/63 - 109s - loss: 0.4158 - accuracy: 0.9089 - val_loss:
    1.4163 - val_accuracy: 0.7395
Epoch 10/25
63/63 - 110s - loss: 0.3992 - accuracy: 0.9109 - val_loss:
    1.5568 - val_accuracy: 0.7194
Epoch 11/25
63/63 - 115s - loss: 0.3342 - accuracy: 0.9129 - val_loss:
    1.6526 - val_accuracy: 0.7034
Epoch 12/25
63/63 - 123s - loss: 0.3676 - accuracy: 0.9084 - val_loss:
    1.6036 - val_accuracy: 0.7355
Epoch 13/25
```

```
63/63 - 125s - loss: 0.2876 - accuracy: 0.9284 - val_loss:
    1.6836 - val_accuracy: 0.7315
Epoch 14/25
63/63 - 131s - loss: 0.3469 - accuracy: 0.9119 - val_loss:
    1.6859 - val_accuracy: 0.7234
Epoch 15/25
63/63 - 126s - loss: 0.3176 - accuracy: 0.9214 - val_loss:
    1.8548 - val_accuracy: 0.7275
Epoch 16/25
63/63 - 118s - loss: 0.3010 - accuracy: 0.9309 - val_loss:
    1.6683 - val_accuracy: 0.7415
Epoch 17/25
63/63 - 133s - loss: 0.3192 - accuracy: 0.9289 - val_loss:
    1.9135 - val_accuracy: 0.7174
Epoch 18/25
63/63 - 124s - loss: 0.3474 - accuracy: 0.9269 - val_loss:
    2.0325 - val_accuracy: 0.7214
Epoch 19/25
63/63 - 121s - loss: 0.3540 - accuracy: 0.9234 - val_loss:
    2.0994 - val_accuracy: 0.7315
Epoch 20/25
63/63 - 127s - loss: 0.3056 - accuracy: 0.9289 - val_loss:
    1.8089 - val_accuracy: 0.7255
Epoch 21/25
63/63 - 133s - loss: 0.2721 - accuracy: 0.9349 - val_loss:
    2.0058 - val_accuracy: 0.7134
Epoch 22/25
63/63 - 115s - loss: 0.3344 - accuracy: 0.9259 - val_loss:
    1.9703 - val_accuracy: 0.7154
Epoch 23/25
63/63 - 132s - loss: 0.2463 - accuracy: 0.9439 - val_loss:
    1.8110 - val_accuracy: 0.7255
Epoch 24/25
63/63 - 131s - loss: 0.3256 - accuracy: 0.9279 - val_loss:
    1.9864 - val_accuracy: 0.7214
Epoch 25/25
63/63 - 118s - loss: 0.3179 - accuracy: 0.9304 - val_loss:
    2.0116 - val_accuracy: 0.7174
```

Transfer learning is already effective in the first training epochs. However, now that the classifier has adapted enough to the new task, it is possible to carry out a second training, with the possibility for all

the layers to modify their weights but only by a small amount. The second training cycle, therefore, involves all the network layers but with a much smaller learning rate.

```
def unfreeze_model(model):
    # We unfreeze the top 20 layers while leaving BatchNorm
   layers frozen
    for layer in model.layers[-20:]:
        if not isinstance(layer, layers.BatchNormalization):
            layer.trainable = True

    optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)
    model.compile(
        optimizer=optimizer, loss="
   sparse_categorical_crossentropy", metrics=["accuracy"]
    )


unfreeze_model(model)

epochs = 20  # @param {type: "slider", min:8, max:50}
hist = model.fit(train_ds, epochs=epochs, validation_data=
   val_ds, verbose=2)
plot_hist(hist)
```

The same script was used for Resnet50 and Mobilenetv2. We will see the results in the next chapter.

Chapter 5

Results

5.1 Accuracy and Inference Time

Here we can see the results for all models, in terms of accuracy and inference times, starting from the Raspberry and proceeding with the same order we saw in chapter 4.

network	accuracy	inference time (ms)
ResNet50	48,58%	670
MobileNetv1	69,51%	128
MobileNetv2	70,96%	116,9
EfficientNetB0 (S)	76,83%	443,3
EfficientNetB1 (M)	78,81%	686,7
EfficientNetB3 (L)	80,92%	1684,9

Table 1: Results on Raspberry Pi

The first comment should be made on the ResNet accuracy. We remember that here the models are fully converted to int8, which can be a problem for models that were initially designed to work in a more precise format, such as ResNet. This is why there is this much difference in accuracy between ResNet and other models, which maintain high levels of accuracy despite quantization. About inference times, it's no surprise to see such slow results on the Raspberry. Since it hasn't

an NPU, all the tensor operations are managed by the CPU, which we know is much less performing than the rest of the devices in this research. However, it's interesting to see how a board so widely used nowadays behaves in such a difficult task.

network	accuracy	inference time (ms)
ResNet50	48,70%	60,8
MobileNetv1	69,44%	4,7
MobileNetv2	70,96%	5,1
EfficientNetB0 (S)	76,85%	9,1
EfficientNetB1 (M)	78,87%	13,3
EfficientNetB3 (L)	80,95%	36

Table 2: Results on Coral USB Accelerator

network	accuracy	inference time (ms)
ResNet50	48,70%	48,9
MobileNetv1	69,44%	3,2
MobileNetv2	70,96%	3,4
EfficientNetB0 (S)	76,85%	5,4
EfficientNetB1 (M)	78,87%	9,2
EfficientNetB3 (L)	80,95%	25,6

Table 3: Results on Coral Dev Board

For the inference time of the two Coral devices, we can immediately note their potential: only a few milliseconds per inference for MobileNets and EfficientNetB0. The other models report slower times, but this is justified by their larger size. Given these results, we can say that the two Coral devices provide enough computing power for complex image classification tasks. This is also quite promising if we would use them for object detection in the future, as they could reach a high number of FPS.

network	accuracy	inference time (ms)
ResNet50	46,15%	23
MobileNetv1	68,89%	11
MobileNetv2	70,69%	12

Table 4: Results on Rock Pi

We can see the same problem as in Coral with the ResNet. Again, due to quantization, the accuracy is quite low, but we notice a significant improvement in inference time. Rock Pi registers the lowest inference time for the ResNet.

network	accuracy	inference time
ResNet50	69,74%	29,1
MobileNetv1	73,34%	18,8
MobileNetv2	74,07%	19,3
EfficientNetB0 (S)	78,93%	29,6
EfficientNetB1 (M)	80,62%	46,2
EfficientNetB3 (L)	83,07%	94,4

Table 5: Results on Jetson Nano

With these results, we can appreciate the advantage of working with floating-point numbers. For the first time, ResNet accuracy reaches nearly 70%, which is in line with most of the previous results. We also notice a slight improvement in all other models, with a maximum in EfficientNetB3, reaching 83% accuracy.

To have a comparison between neural networks in their original form and their modified versions, on x86 and with the use of Keras, we carried out some accuracy tests with respect to ImageNetv2 dataset. Here we can compare all the already seen accuracy values with the baseline.

network	baseline	Raspberry	Coral USB	Coral Board	Rock Pi	Jetson
ResNet50	69,71%	48,58%	48,70%	48,07%	46,15%	69,74%
MobileNetv1	70,69%	69,51%	69,44%	69,44%	68,89%	73,34%
MobileNetv2	70,46%	70,96%	70,96%	70,96%	70,69%	74,07%
EfficientNetB0 (S)	74,78%	76,83%	76,85%	76,85%		78,93%
EfficientNetB1 (M)	76,81%	78,81%	78,87%	78,87%		80,62%
EfficientNetB3 (L)	81,00%	80,92%	80,95%	80,95%		83,07%

Table 6: Comparison with the baseline

We already knew that ResNet gets penalized for switching to int8, here we can see the amount of this loss is and how the Jetson Nano is the only board capable of replicating the original results. Looking at the results obtained by the other networks, we can see that they don't suffer from the quantization, an indication that these are networks originally designed for the eventual use in int8 format.

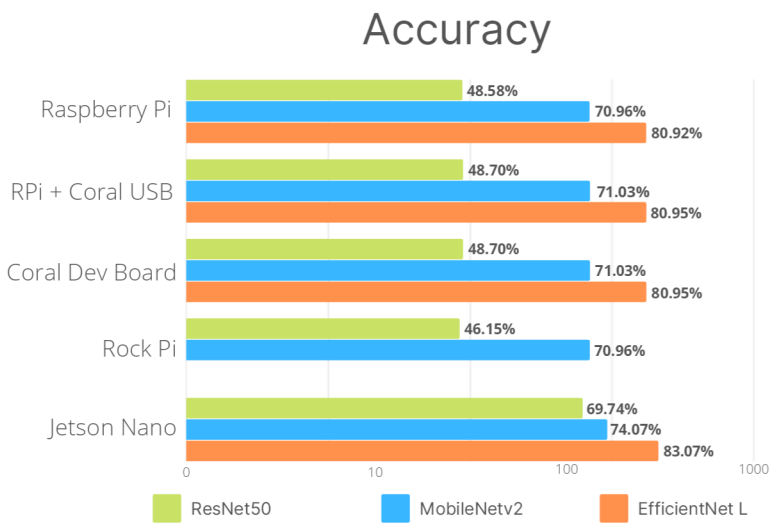

Figure 16: Accuracy chart

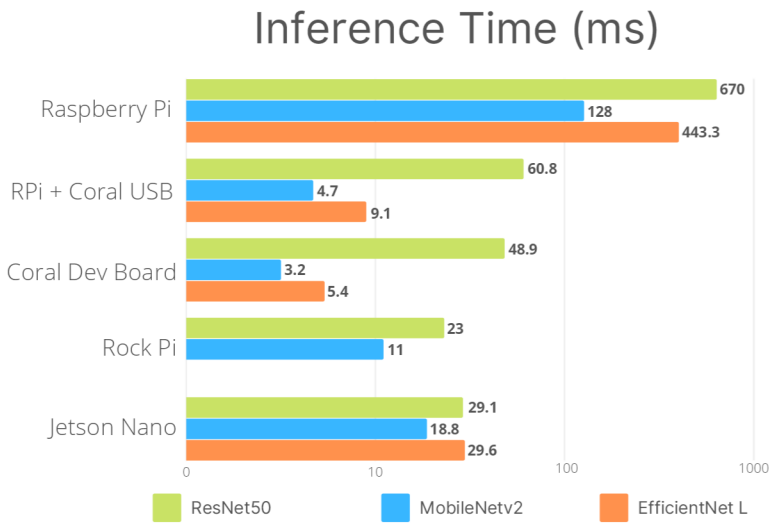

Figure 17: Inference time chart

Here we can see the results for inference time on a logarithmic scale. Coral devices are generally the fastest, while in the last place, apart from the Raspberry, we find the Jetson Nano. As we mentioned earlier, this is what we expected when looking at the specifications declared by the manufacturers.

5.2 Power Consumption

The following results will refer to the average consumption of the SBCs during the inference operations.

usage	Raspberry	Coral USB	Coral Board	Rock Pi	Jetson
idle	2.5	3.9(1)	2.8	2.5	1.1
ResNet50	5.4	5.1(1.2)	4.2	3.3	4.2
MobileNetv1	5.4	5.2(1.1)	4.7	3.3	3.7
MobileNetv2	5.4	5.3(1.1)	4.7	3.3	3.9
EfficientNetB0 (S)	5.4	5.4(1.2)	4.8		5.5
EfficientNetB1 (M)	5.4	5.4(1.3)	4.8		5.7
EfficientNetB3 (L)	5.5	5.6(1.5)	4.8		6.2

Table 7: Power consumption (W)

Let's start by talking about the raspberry. Since the beginning of this research, it has been the device with the least expectations, given the absence of the NPU, and as we have already seen it is in last place in terms of speed and average in terms of accuracy. For the power consumption, the only field in which the raspberry could have a chance to stand out compared to the other boards, we find results quite far from optimality, confirming the inefficiency of this device in this specific context. In this regard, it is very interesting to note that power consumption is lower during tests with the USB accelerator connected to it. This happens because raspberry CPU requires a lot of effort for doing so many operations, which results in high power consumption. Raspberry is generally more suitable for smaller and easily manageable neural networks. The lower power consumption given by the use of the accelerator demonstrates its efficiency in the field for which it was designed.
A small comparison could be made between the Coral Dev Board and Jetson Nano: the first one is a device that gave very similar results, which depend very little on the type of neural network that is used, while Jetson Nano consumes less with small networks like MobileNet but more with larger networks. It seems that in the Jetson Nano the

power consumption is strictly related to the neural network. The reason for that may regard the board's architecture and how the operations management system works, which may be different from the Dev Board. The last device to discuss is Rock Pi. Its results are the most encouraging, as it is the board with the lowest average consumption for all the three neural networks we have been able to test. We can't know whether the same trend would be followed also in the EfficientNet case, but for the moment we can say that we have excellent results.

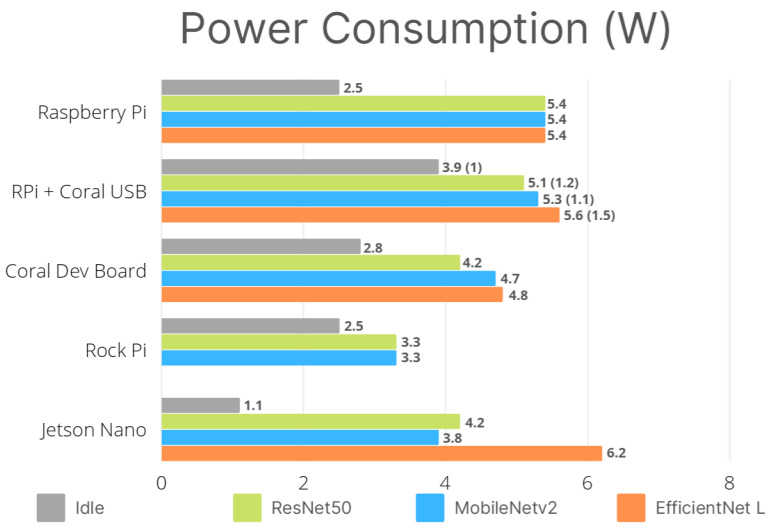

Figure 18: Power consumption chart

The data just seen refer to the average power during the inference. Another fact that we can show is the total energy consumption. The reason why we show the next table is to have a measure that includes inference time and power consumption at the same time, to understand in terms of energy whether it is better to opt for a fast but consuming board or another with opposite characteristics. Please note that the results refer only to the consumption during inference,so without counting all the other operations in between.

network	Raspberry	Coral USB	Coral Board	Rock Pi	Jetson
ResNet50	3.638	310	206	76	122
MobileNetv1	691	25	15	36	70
MobileNetv2	631	27	16	39	75
EfficientNetB0 (S)	2.415	49	26		162
EfficientNetB1 (M)	3.741	72	44		264
EfficientNetB3 (L)	9.200	202	123		589

Table 8: Comsumption during inference (mJ)

5.3 Rings Training

network	training accuracy	validation accuracy
ResNet50	97,20%	61,92%
MobileNetv2	97,65%	65,93%
EfficientNetB0 (S)	98,40%	75,95%

Table 9: Training results

Although we have tried all types of Networks used for the benchmark, we noticed after the first tests that the EfficientNet achieved slightly better results, which is why more tests were done on this network in particular. We tried more changes on the number of epochs, learning rate, and preprocessing, which led to more small improvements. The following results are not a comparison between the networks, instead, there are used as a report of the current state of the training, where we

show the best results obtained so far. We cannot say that the study on this dataset is finished, as many tests can be done to deal with the most evident problem which is overfitting. The difference between training accuracy and validation accuracy in the best result is 22.45%, which cannot be ignored. Usually, we always try to find a balance between the two accuracy, which in this case has not been achieved, so we hope to further improve these results in the future.

Chapter 6

Conclusions and Future Work

There are several conclusions that we can make from this research:

- The data gathered with these experiments prove to be useful in choosing a configuration based on different constraints. It is very interesting that for each of the 3 KPIs we have chosen we identify a different board as the best. Coral devices are the fastest, in fact even in our experiments we notice a certain consistency with the declared specifications, reaching the best inference times. NVIDIA Jetson Nano is the board with the highest accuracy, thanks to the use of networks with high precision, while for the power consumption, if we consider the values during inference, the best choice is Rock Pi. Since all these devices are recent, we have not found any critical issues in the boards, for example, a Jetson Nano offers viable performance even for speed and consumption, and the same thing also applies to the other devices. Therefore, for the image classification task with the use of SBCs, Coral, Jetson, and Rock Pi can all be taken into consideration, and the final choice in each situation will depend on a 3-way trade-off that must be analyzed case by case.

- The ResNet50 is a neural network widely used for various projects. However, since was not designed for the deployment on edge devices, which in most cases requires model quantization, it has few applications in the IoT field. However, we found valid alternatives, including MobileNet and EfficientNet, able to work in int8 and generally more versatile.

- It is important to note that all tests were performed on a very difficult task. Not all image classification scenarios are of this type: in many cases, we work with datasets with a fewer number of classes, all distinguishable from each other. Therefore, using the most accurate board, in our case the Jetson, is not always the best solution. Some tasks do not need a neural network too complex or fp32 precision to reach high accuracy values, and in those situations, the choice of the board can be more related to speed. Power consumption could also play a fundamental role in some situations, especially if we want to use the device without a direct power source. It is therefore essential to have a clear idea of the problem and the tools in our possession, to make the best choice in each situation.

We have come to the end of this thesis, where the last thing left to ask is if this research can be improved, or what we would have done if we had had more time available. We will then describe the possible improvements, which could be made in the near future.

As in any other field, the state of the art of this project is in constant evolution, which is why any future work has to involve the exploration of new SBCs or new convolutional neural networks, also concerning the results we have achieved these last months. Extending the analysis to other tasks would be another great idea. Object detection, semantic segmentation, and speech recognition could be some examples. It is clear that to have a complete picture of the IoT world takes much more time, and because of its importance, we will certainly see more works by the research community which will further extend our knowledge.

We also hope that someday we will deploy EfficientNet on Rock Pi: whether the issue was about compatibility or not, we are confident that

we will get the answers we are searching for.

A more precise measurement of the average power is possible with the appropriate tools and enough time to invest on. An idea could be to use a device that can communicate with a host to transmit the readings data, so we could be able to plot graphs regarding the power trend or even get the confidence intervals for the average power.

For the training on the rings dataset, with more time we would have tried to eliminate overfitting. We hope that by exploring other techniques, such as different data augmentation, a different data split between the training set and the validation set, or even expanding the dataset with more, the overfitting problem will one day be solved.

As for me, this project signs the end of a journey that was five years and a half long, so besides the project future work i also asked myself how *my* future will be.
I've learned a lot in the last years, about the world surrounding me, about all the things I've always wondered how they works, if they can be improved, what my contribution can be, and all the other things that make you an engineer.
I've also learned a lot about myself, my strengths, weaknesses, fears, hopes and dreams.
I've experienced the failure of an exam gone wrong and the joy of one that went as expected, I've experienced the 8 to 7 study sessions and the relax days after a well done exam session, and now i know better what is the *cost* needed to reach an objective and how it feels to achieve it.
I'm closing an important chapter of my life, but i really hope to never forget what brought me to this point. May this experience be a guiding light for all that comes next to my master degree proclamation.
To conclude, besides all the acknowledgments, I'd like to dedicate this work to my future me, hoping he will read again this message after reaching our next milestone.

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge Computing: Vision and Challenges». In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on pp. 2, 23).

[2] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. «Challenges and Opportunities in Edge Computing». In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18 (cit. on pp. 2, 23).

[3] *Raspberry Pi*. https://www.raspberrypi.org/ (cit. on p. 2).

[4] *Coral USB Accelerator*. https://coral.ai/products/accelerator (cit. on p. 2).

[5] *Coral Dev Board*. https://coral.ai/products/dev-board (cit. on p. 2).

[6] *Rock Pi*. https://wiki.radxa.com/RockpiN10 (cit. on p. 2).

[7] *NVIDIA Jetson Nano*. https://developer.nvidia.com/embedded/jetson-nano-developer-kit (cit. on p. 2).

[8] *ImageNet Website*. https://www.image-net.org/ (cit. on p. 3).

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «ImageNet: A large-scale hierarchical image database». In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848 (cit. on pp. 3, 14).

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV] (cit. on pp. 3, 10, 11).

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV] (cit. on pp. 3, 10, 12).

[12] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG] (cit. on pp. 3, 10, 12, 13).

[13] *Keras library*. https://keras.io/ (cit. on pp. 3, 15).

[14] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC] (cit. on pp. 3, 15).

[15] Charles R. Harris et al. «Array programming with NumPy». In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. URL: http://dx.doi.org/10.1038/s41586-020-2649-2 (cit. on pp. 3, 15).

[16] François Chollet. *Deep Learning with Python*. Manning Publications, 2018 (cit. on p. 5).

[17] *Venn diagram source*. https://levity.ai/blog/difference-machine-learning-deep-learning (cit. on p. 6).

[18] *ANN image source*. https://www.tutorialspoint.com/machine_learning/machine_learning_artificial_neural_networks.htm (cit. on p. 7).

[19] *Convolution image source*. https://medium.com/ai-salon/understanding-deep-self-attention-mechanism-in-convolution-neural-networks-e8f9c01cb251 (cit. on p. 9).

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf (cit. on p. 10).

[21] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. arXiv: 1801.04381 [cs.CV] (cit. on p. 10).

[22] *Wordnet*. https://wordnet.princeton.edu/ (cit. on p. 14).

[23] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. *Do ImageNet Classifiers Generalize to ImageNet?* 2019. arXiv: 1902.10811 [cs.CV] (cit. on p. 15).

[24] *Imagenetv2 repository*. https://github.com/modestyachts/ImageNetV2 (cit. on p. 15).

[25] *pycoral repository*. https://github.com/google-coral/pycoral (cit. on pp. 15, 16, 31).

[26] *rknn-toolkit repository*. https://github.com/rockchip-linux/rknn-toolkit (cit. on pp. 15, 16, 36).

[27] *TensorRT repository*. https://developer.nvidia.com/tensorrt (cit. on p. 15).

[28] *GitHub website*. https://github.com/ (cit. on p. 16).

[29] *jetson-inference repository*. https://github.com/dusty-nv/jetson-inference (cit. on pp. 16, 43).

[30] *repository for EfficientNet and MobileNet benchmark*. https://github.com/NVIDIA/TensorRT/tree/main/samples/python/efficientnet (cit. on pp. 16, 43).

[31] *Transfer Learning example*. https://github.com/keras-team/keras-io/blob/master/examples/vision/image_classification_efficientnet_fine_tuning.py (cit. on pp. 16, 53).

[32] *Colab notebook for edgetpu conversion*. https://colab.research.google.com/github/google-coral/tutorials/blob/master/compile_for_edgetpu.ipynb (cit. on pp. 16, 30).

[33] Mi Zhang, Faen Zhang, Nicholas D. Lane, Yuanchao Shu, Xiao Zeng, Biyi Fang, Shen Yan, and Hui Xu. *Deep Learning in the Era of Edge Computing: Challenges and Opportunities*. 2020. arXiv: 2010.08861 [cs.LG] (cit. on p. 23).

[34] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. «RIoTBench: An IoT benchmark for distributed stream processing systems». In: *Concurrency and Computation: Practice and Experience* 29.21 (Oct. 2017), e4257. ISSN: 1532-0626. DOI: 10.1002/cpe.4257. URL: http://dx.doi.org/10.1002/cpe.4257 (cit. on p. 23).

[35] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. «IoTAbench: An Internet of Things Analytics Benchmark». In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 133–144. ISBN: 9781450332484. DOI: 10.1145/2668930.2688055. URL: https://doi.org/10.1145/2668930.2688055 (cit. on p. 23).

[36] Yihui Ren, Shinjae Yoo, and Adolfy Hoisie. *Performance Analysis of Deep Learning Workloads on Leading-edge Systems*. 2019. arXiv: 1905.08764 [cs.PF] (cit. on p. 24).

[37] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. «An Early Resource Characterization of Deep Learning on Wearables, Smartphones and Internet-of-Things Devices». In: *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*. IoT-App '15. Seoul, South Korea: Association for Computing Machinery, 2015, pp. 7–12. ISBN: 9781450338387. DOI: 10.1145/2820975.2820980. URL: https://doi.org/10.1145/2820975.2820980 (cit. on p. 24).

[38] Stephan Patrick Baller, Anshul Jindal, Mohak Chadha, and Michael Gerndt. *DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices*. 2021. arXiv: 2108.09457 [cs.AI] (cit. on p. 24).

[39] Mattia Antonini, Tran Huy Vu, Chulhong Min, Alessandro Montanari, Akhil Mathur, and Fahim Kawsar. «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, 2019, pp. 49–55. ISBN: 9781450370134. DOI: 10.1145/3363347.3363363. URL: https://doi.org/10.1145/3363347.3363363 (cit. on pp. 24, 51).

[40] Pilsung KANG and Jongmin JO. «Benchmarking Modern Edge Devices for AI Applications». In: *IEICE Transactions on Information and Systems* E104.D (Mar. 2021), pp. 394–403. DOI: 10.1587/transinf.2020EDP7160 (cit. on p. 24).

[41] *Coral USB Accelerator setup*. https://coral.ai/docs/accelerator/get-started/ (cit. on p. 30).

[42] *Coral Dev Board setup*. https://coral.ai/docs/dev-board/get-started/ (cit. on p. 30).

[43] *Coral models*. https://github.com/google-coral/test_data/tree/104342d2d3480b3e66203073dac24f4e2dbb4c41 (cit. on p. 30).

[44] *Rock Pi guide*. https://forum.radxa.com/t/guide-how-to-get-the-rock-pi-n10-up-and-running-with-npu-inference/4632 (cit. on p. 35).

[45] *Radxa Forum*. https://forum.radxa.com/ (cit. on p. 39).

[46] *PuTTy*. https://www.putty.org/ (cit. on p. 43).

[47] *Jetpack*. https://docs.nvidia.com/jetson/jetpack/index.html (cit. on p. 43).

[48] *CUDA*. https://developer.nvidia.com/cuda-toolkit (cit. on p. 43).

[49] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. *A Survey on Deep Transfer Learning*. 2018. arXiv: `1808.01974 [cs.LG]` (cit. on p. 52).