Implementation and Performance Evaluation of a Semantic Image Segmentation System on a Mobile Device

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Implementación y Evaluación de Rendimiento de un Sistema de Segmentación Semántica de Imágenes sobre un Dispositivo Móvil

Trabajo Fin de Grado
Curso 2020-2021

Autora
Esther Carreño Alocén

directores
Luis Piñuel Moreno
Francisco D. Igual

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid
Madrid, septiembre de 2021

ACKNOWLEDGEMENTS

To the project directors Luis Piñuel Moreno and Francisco D. Igual, for all their trust and support. To my good friend Martín Bárez, who has always been there for me. To my boyfriend, for helping me not to fall into the quicksand of my mind. [55]

Abstract

There has been a growth of interest in semantic segmentation in recent times, and its employment in tasks such as autonomous driving, medical diagnosis or video surveillance is now crucial. The training and inference processes in Deep Neural Networks (DNNs) are performed in data centres, which causes unbearable latency. Edge Computing is a response to this limitation. Nevertheless, it is restricted by the computing power and energy consumption in devices.

This project proposes the implementation of algorithms for semantic segmentation of images using DeepLab and TensorFlow as a basis, along with its adaptation and throughput evaluation in terms of response time in a mobile device among different semantic segmentation models.

A Raspberry Pi was used along with a Coral USB Accelerator by Google, which provides an Edge TPU to accelerate inference in Machine Learning by quantizing the models. The final goal is to prove an efficient implementation in this low energy consumption architecture.

Keywords

Semantic segmentation, DNNs, Edge Computing, DeepLab, Coral, Energy Efficiency, Latency, Raspberry Pi

Resumen

En los últimos tiempos ha habido un aumento en el interés por la segmentación semántica, y su uso en tareas como la conducción autónoma, el diagnóstico médico o la videovigilancia es crucial. Los procesos de entrenamiento e inferencia en Redes Neuronales Profundas (DNNs) se realizan en centros de datos, lo que causa una latencia insostenible. El Edge Computing es una respuesta a esta limitación, pero está restringido por la potencia computacional y el consumo de energía de los dispositivos.

Este proyecto propone la implementación de algoritmos de segmentación semántica de imágenes usando como base DeepLab y TensorFlow, además de su adaptación y evaluación de rendimiento según el tiempo de respuesta en dispositivos móviles entre diferentes modelos de segmentación semántica.

Para ello se ha utilizado una Raspberry Pi y se ha optado por el acelerador USB Coral, de Google, que ofrece un Edge TPU para acelerar la inferencia en Machine Learning mediante la cuantización. El objetivo final es demostrar que una implementación eficiente en una arquitectura de bajo consumo energético es posible.

Palabras clave

Segmentación semántica, DNNs, Edge Computing, DeepLab, Coral, Eficiencia Energética, Latencia, Raspberry Pi

Content Index

Acknowledgements III

Abstract IV

Resumen V

Content Index VII

Figure Index XI

Table Index XIII

Capítulo 1 - Introduction 1

1.1 Motivation 1

1.1.1 Semantic segmentation 1

1.2 Goals 3

1.3 Project plan 3

Capítulo 2 - State of art 5

2.1 Convolutional Neural Networks 6

2.2 Transfer Learning 9

2.2.1 Transfer Learning in Edge-TPU 9

2.3 Quantization 10

2.3.1 How to quantize 10

2.3.3 Risks of quantization 12

Capítulo 3 - Hardware devices and architectures used 13

3.1 Laptop i7 NVIDIA® GeForce® GTX 13

3.2 Virtual Machines 13

3.3 Raspberry Pi 4 Model B 8GB RAM 13

3.4 USB Accelerator Google Coral 15

3.4.1 Edge TPU 16
Capítulo 4 - Frameworks used 17
4.1 TensorFlow 17
4.2 TensorFlow Lite 17
4.2.1 Post-training quantization 18
4.2.1.1 Full integer post-training quantization 19
4.2.1.2 TensorFlow Lite converter 20
4.3 DeepLab 21
4.3.1 How does DeepLab work 21
4.3.1.1 Spatial pyramid pooling 22
4.3.1.2 Atrous Convolutions 22
4.3.1.3 Depthwise Separable Convolutions 23
4.3.2 Datasets 23
4.3.3 Model Zoo 24
4.3.4 Evaluation metrics 26
4.3.5 Advantages of DeepLab 26
4.4 OpenCV 26
Capítulo 5 - Implementation and experimentation 27
5.1 DeepLab local installation 27
5.1.1 Laptop installation 27
5.1.1.1 Training 28
5.1.1.2 Evaluation 30
5.1.1.3 Visualization 31
5.1.1.4 Script (draft) 32
5.2 Virtual Machine 34

5.2.1 Virtual Machine configuration 35
5.2.2 Updated script 35
5.2.3 Measured times for each model 36
5.3 DeepLab in Raspberry Pi 4 37
5.3.1 Raspberry Pi configuration 37
5.3.2 Measured times for each model 38
5.4 Coral USB Accelerator in Raspberry Pi 4 44
5.4.1 Coral installation 44
5.4.2 How to run a model on the Edge TPU 44
5.4.3 How to run TFLite object detection models on the Raspberry Pi 45
5.4.4 How to run Edge TPU object detection models on the Raspberry Pi using Coral USB Accelerator 46
5.4.5 How to quantize a model to run it on Edge TPU 48
Capítulo 6 - Results Comparison and Analysis 53
6.1 Laptop vs. Virtual Machine execution times 53
6.2 Virtual Machine image input vs. Virtual Machine webcam input 53
6.3 Virtual Machine vs. Raspberry Pi 54
6.4 Raspberry Pi: regular model vs. Edge TPU model 55
Capítulo 7 - Conclusions and future work 57
Bibliografía 59
Apéndice A - Configuration Guidelines 65

Figure Index

Figure 1-1. Example of semantic image segmentation

Figure 2-1. Example of a CNN sequence to classify handwritten numbers

Figure 2-2. 4x4x3 RGB Image

Figure 2-3. Convolutioning process

Figure 2-4. Quantization technique decision tree

Figure 3-1. Raspberry Pi 4 Model B architecture.

Figure 3-2. USB Accelerator Google Coral.

Figure 3-3. Two Edge TPU chips on a US penny

Figure 4-1. Workflow to create a model for Edge TPU

Figure 4-2. TFLite conversion diagram

Figure 4-3. Segmentation result example on Flickr image

Figure 4-4. Spatial pyramid pooling

Figure 4-5. Example mobilenetv2_coco_voc_trainaug

Figure 4-6. Example xception65_coco_voc_trainval

Figure 5-1. Console output when running model_test.py

Figure 5-2. Recommended Directory structure for Training and Evaluation

Figure 5-3. Command used to run train.py

Figure 5-4. Terminal output when training DeepLab using the official repository (train.py)

Figure 5-5. Command used to run eval.py

Figure 5-6. Command used to run vis.py

Figure 5-7. Segmentation results after running vis.py

Figure 5-8. mobilenetv2_coco_voctrainaug segmentation laptop output example

Figure 5-9. mobilenetv2_coco_voctrainval segmentation laptop output example

Figure 5-10. xception_coco_voctrainaug segmentation laptop output example

Figure 5-11. xception_coco_voctrainval segmentation laptop output example

Figure 5-12. Raspberry Pi installation interface

Figure 5-13. RasPi Unclear input, xception_coco_voctrainval

Figure 5-14. RasPi Unclear input, xception_coco_voctrainaug

Figure 5-15. RasPi Unclear input, mobilenetv2_coco_voctrainval

Figure 5-16. RasPi Unclear input, alternative mobilenetv2_coco_voctrainval

Figure 5-17. RasPi Unclear input, mobilenetv2_coco_voctrainaug

Figure 5-18. RasPi Clear input, xception_coco_voctrainval

Figure 5-19. RasPi Clear input, xception_coco_voctrainaug

Figure 5-20. RasPi Clear input, mobilenetv2_coco_voctrainval

Figure 5-21. RasPi Clear input, mobilenetv2_coco_voctrainaug

Figure 5-22. RasPi Clear input, alternative mobilenetv2_coco_voctrainaug

Figure 5-23. RasPi Several objects, xception_coco_voctrainval

Figure 5-24. RasPi Several objects, xception_coco_voctrainaug

Figure 5-25. RasPi Several objects, mobilenetv2_coco_voctrainval

Figure 5-26. RasPi Several objects, mobilenetv2_coco_voctrainaug

Figure 5-27. Edge TPU inference results

Figure 5-28. TFLite object detection models on a Raspberry Pi

Figure 5-29. Edge TPU object detection models on a Raspberry Pi using Coral

Figure 5-30. Terminal output when converting the TF model to a TF Lite model

Figure 5-31. Terminal output when compiling the TF Lite model to Edge TPU

Figure 5-32. Inference result of dummy_edgetpu.tflite model with input image

Figure 5-33. Inference result of smarty_edgetpu.tflite model with input image

Figure 5-34. Inference result of smarty_edgetpu.tflite model with webcam input

TABLE INDEX

Table 1-1. Gantt diagram with an overview the tasks done

Table 2-1. Compared inference times of DNNs models for edge and cloud computing.

Table 2-2. Quantization methods and their performance in TensorFlow Lite.

Table 4-1. Post-training quantization techniques

Table 4-2. Computation complexity (in terms of Multiply-Adds and CPU Runtime) and segmentation performance (in terms of mIOU) on the PASCAL VOC val or test set

Table 5-1. Execution time of semantic segmentation (in seconds) using different models

Table 5-2. Execution time of semantic segmentation (in seconds) using different models in the Virtual Machine

Table 5-3. Semantic segmentation execution times (in seconds) of webcam images in the Virtual Machine

Table 5-4. Semantic segmentation execution times (in seconds) of webcam images in the Raspberry Pi 4

Table 6-1. Inference time comparison (in seconds) between Virtual Machine and Raspberry Pi

Table 6-2. Time comparison (in seconds) between regular and EdgeTPU semantic segmentation model in Raspberry Pi

Capítulo 1 - Introduction

1.1 Motivation

In the last few years there has been a major breakthrough in Artificial Intelligence. This great stride was possible thanks to the progress of processors' computing capacity.

Nowadays, there are a large amount of services that have been automated, and the tendency to avoid human interaction for different tasks is increasing to the point where autonomous processes are rapidly expanding. For that purpose, we use Deep Learning in order that the computer behaves like a human during the execution of an assignment. In particular, we use Convolutional Neural Network (CNN) because they are designed to process image pixel data.

1.1.1 Semantic segmentation

Semantic segmentation is a part of computer vision, it can be described as the assignment of a label (which refers to a concept, for instance a car or a person) to each pixel in an image according to the object that has been detected. It has become a key process necessary for multiple tasks, such as autonomous driving (by detecting signs and obstacles), medical diagnosis automated through images (this could be used for early detection of tumours), crowd management and people counting, and video surveillance, among others. To this effect, DeepLab, a semantic segmentation model developed by Google and based on Tensorflow, is noteworthy.

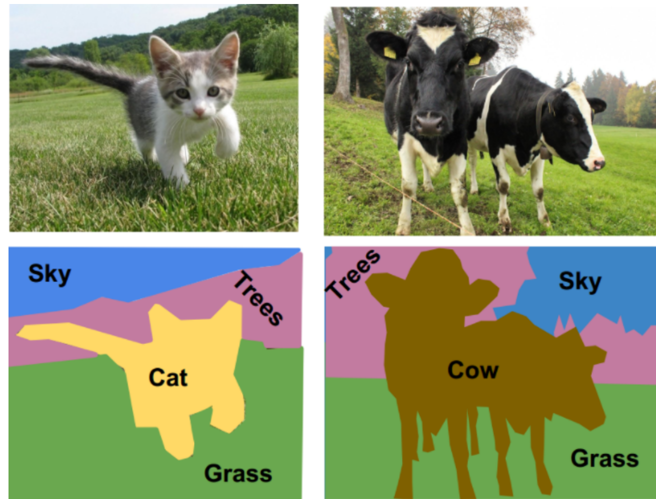


Figure 1-1. Example of semantic image segmentation. [18]

As the *Figure 1-1* shows, semantic segmentation labels each pixel in the image with a category label, however it does not differentiate instances. This means that it does not separate the objects of the same category. In this example, we can see that both cows in the image are considered as a whole unit of pixels that correspond to the cow category.

There has been a growth of interest in this field, motivated by the revival of the Deep Neural Networks (DNNs) and the improvement of processors' computing capacity. However, it has to be taken into consideration that the training and inference processes in DNNs are performed in data centres (in cloud), which often leads to an unbearable response time or latency. In response to this limitation, Edge Computing is being used, which consists in performing the operations near the device which needs the calculations. This approach reduces the latency, but it is limited by the computing power as well as the energy consumption in these devices.

Note that inference refers to the process of making predictions on unseen data using trained DNNs model.

1.2 Goals

This project aims to study a neural network model and to compare its metrics, like precision and latency, depending on the trained models used, their datasets and the architecture by creating a semantic segmentation application.

The final goal is to prove an efficient implementation in a low energy consumption architecture.

1.3 Project plan

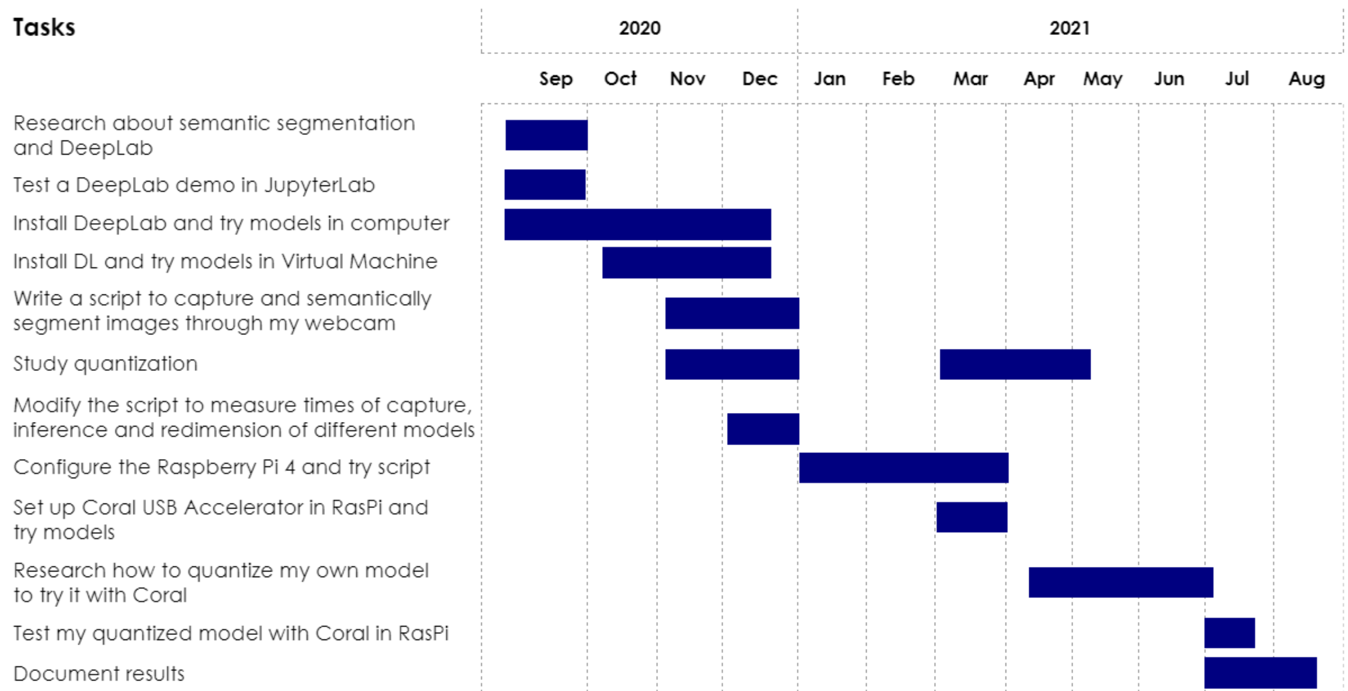


Table 1-1. Gantt diagram with an overview the tasks done

Capítulo 2 - State of art

Training and inference processes in DNNs are usually performed in data centres. Although some cloud services, such as AWS, provide inference services like Amazon Polly, Rekognition and Lex, it is not enough. When devices send data to the cloud for processing, the reception of the final results is totally dependent on the congestion as well as the latency of internet connection. Moreover, there is a lack of efficiency in regards to cost and energy when streaming dense information, like images and video. It should be noted that when working with real-time applications, results need to be returned soon to maintain a positive user experience and to meet the requirements of the application. Because of this, an adequate inference latency is crucial. This means that the faster the prediction is, the higher the number of predictions per time unit we get is, and hence a generally reduced cost. [23]

In response to these limitations of cloud, Edge Computing comes into play; it consists in performing the operations near the device which needs those calculations. This approach reduces the latency, but it is limited by the computing power as well as the energy consumption in these devices.

DNN	Model Size (MB)	GPU Inference (ms)		CPU Inference (ms)	
		Cloud	OnPrem	Cloud	OnPrem
Faster RCNN ResNet 101	196.5	1517	114	1615	676
SSD MobileNet V1	29.1	340	77	459	54
SSD MobileNet V2	69.7	354	75	719	65
SSD Inception V2	102	366	80	624	69
Faster RCNN Inception V2	57.2	619	89	726	221
Mask RCNN Inception V2	67.1	789	97	801	321
DeepLab V3	23	925	70	1371	272
DeepLab V3 Xception	447	3124	147	3061	1585

Table 2-1. Compared inference times of DNNs models for edge and cloud computing. [23]

Table 2-1 shows a comparison between the inference times that we can find of different DNNs models for both cloud and on-premise (edge) computing. From this information it can be gathered that there is a balance between accuracy and latency. In addition, it also compares GPU and CPU. Later on in this project TPU will also be discussed, as the USB Accelerator Google Coral provides an Edge TPU coprocessor for the Raspberry Pi 4 Model B, which will reduce the latency even more.

2.1 Convolutional Neural Networks

Computer Vision (CV) is an area of Artificial Intelligence that imitates the human vision system, in such a manner that computers can identify and process items in images and videos just like humans do. [25]

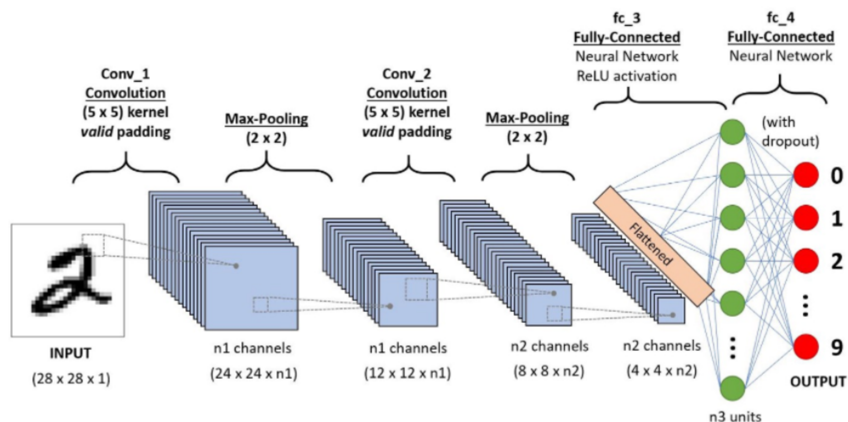


Figure 2-1. Example of a CNN sequence to classify handwritten numbers. [36]

Convolutional Neural Networks (CNNs) are Deep Learning algorithms used in the field of CV. CNNs use an image as an input and give a value to different the objects in that image in order to distinguish all of them. This kind of network is equivalent to the neuron connections in the human brain. [54]

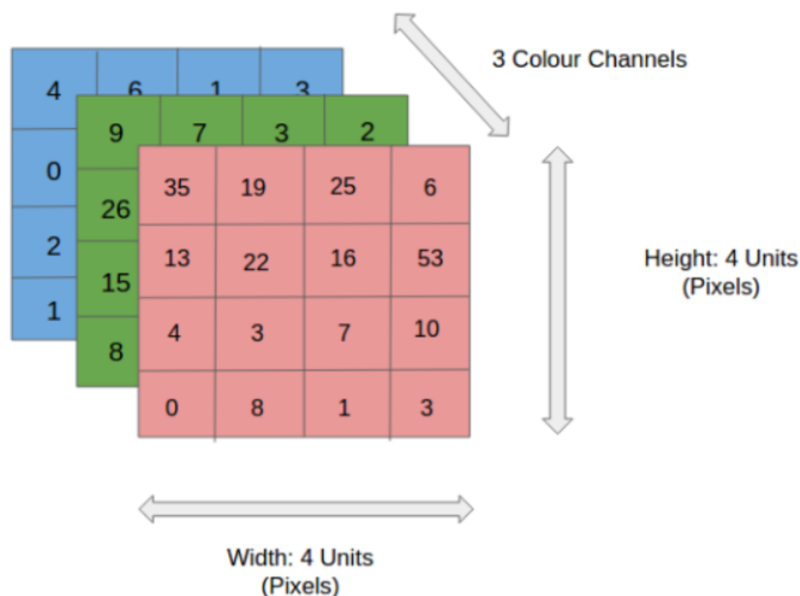


Figure 2-2. 4x4x3 RGB Image. [36]

Figure 2-2 represents the three color planes of a RGB image. In this case, it is only 4x4 pixels, but even when the image has bigger dimensions, the CNN reduces the processing complexity while keeping the necessary information for a good prediction.

In order to implement the first convolution operation in the *convolution layer*, for a 5x5x1 input image there is a Filter (K), which is selected to be 3x3x1.

K = $(101,\ 010,\ 101\ )$

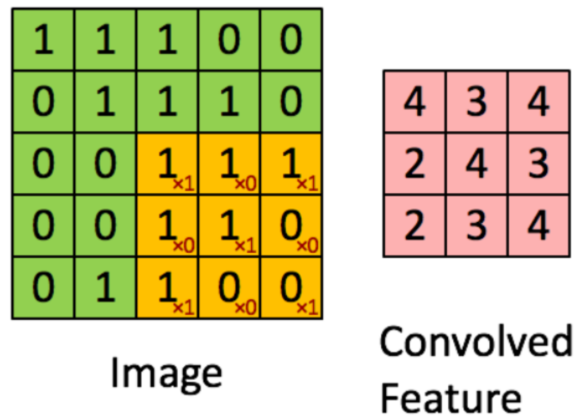


Figure 2-3. Convolutioning process. [36]

Then, the filter shifts 9 times performing a matrix multiplication by the corresponding portion (P) of the image. With the convolution operation it is aimed to extract high-level attributes from the image.

After the convolutional layer, the *pooling layer* reduces as well the size of the image. This way, the processing of the data does not need to employ such a high computational power. Max Pooling is the best type of pooling, given that it returns the maximum value from P and it is a noise suppressant, which means that it rejects the noisy data that does not provide any useful information.

As stated in [36], "by adding a Fully-Connected layer the network learns non-linear combinations of high-level features". Then the image matrix is flattened into a column vector, which is used as an input for a feed-forward neural network and backpropagation is practiced in each iteration of the training process to compute the gradient and loss function.

Finally, the features of the images are classified using the Softmax Classification technique [45] by filtering the values which are below a maximum established value. [36]

2.2 Transfer Learning

Transfer Learning (sometimes also called "fine tuning") focuses on applying the knowledge previously obtained for one task to solve similar problems. [37] In this way, it is not necessary to learn everything from scratch, which would consume time and computing resources.

Hence, the key motivation of using transfer learning is that getting a large quantity of labeled data is necessary for models which solve complex tasks, but labeling that data can be hard in regards to the time and labor employed. Moreover, transfer learning may be highly needed in case the data can be readily outdated because the previously obtained labeled data may not follow the same distribution later. [29]

In semantic segmentation, and generally in computer vision, pre-trained models based on big CNNs are used in transfer learning. [24]

To sum up, we can use a model that is already trained to perform extra training using a smaller training dataset to teach the model new classifications.

2.2.1 Transfer Learning in Edge-TPU

Using transfer learning, it is possible to retrain an existing model compatible with Edge TPU. This can be done by two methods, which are explained at [44]:

- "Retraining the whole model by adjusting the weights across the whole network."
- "Removing the final layer that performs classification, and training a new layer on top that recognizes your new classes. Once you're happy with the model's performance, simply convert it to TensorFlow Lite and then compile it for the Edge TPU. This will be explained later in the work."

2.3 Quantization

As explained previously, NNs employ high computational costs and consume a lot of memory. Because of this, it is important to optimize its training and inference. Nowadays, more models move from servers to edge because of the advantages that edge offers regarding latency. When running models on the edge, network optimization is even more crucial, taking into account their limitations in computing power. One technique for reducing the complexity of CNNs is *quantization*.

The main idea of *quantization* is to convert floating point weights and inputs into the nearest integers. This helps to consume less memory and may lead to faster calculations depending on the hardware. In other words, "it makes the model smaller and faster". [32] Even if the 8-bit integer depiction can be less precise, the inference accuracy of the NN is not remarkably compromised. [31]

There are several ways of representing floating points and integers, but *float32* and *int8* (which ranges from -128 to 127) are the most common for *quantization*. [11]

Although the calculation speed depends on the hardware used, in general, *int8* is typically faster than *float32*. Nevertheless, it must be taken into account that *float32* is used by default for training and inference for NNs. [3]

2.3.1 How to quantize

We can perform quantization through matrix multiplications. This procedure is an approximation; hence, we lose information in the process. However, this may be acceptable sometimes. [11]

There are two main ways to perform quantization:

- *Post-training quantization:* It consists in training the model using the default float32 weights and inputs, and later quantizing the weights. It is easy to perform, but the disadvantage is that it can lead to accuracy loss.
- *Quantization-aware training:* The weights are quantized during the training phase. int8 quantization offers better results, but it is a more complex option.

Technique	Data requirements	Size reduction	Accuracy	Supported hardware
Post-training float16 quantization	No data	Up to 50%	Insignificant accuracy loss	CPU, GPU
Post-training dynamic range quantization	No data	Up to 75%	Accuracy loss	CPU, GPU (Android)
Post-training integer quantization	Unlabelled representative sample	Up to 75%	Smaller accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Quantization-aware training	Labelled training data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP

Table 2-2. Quantization methods and their performance in TensorFlow Lite. [26]

As we can see in *Table 2.2*, there are multiple quantization techniques available for TensorFlow Lite.

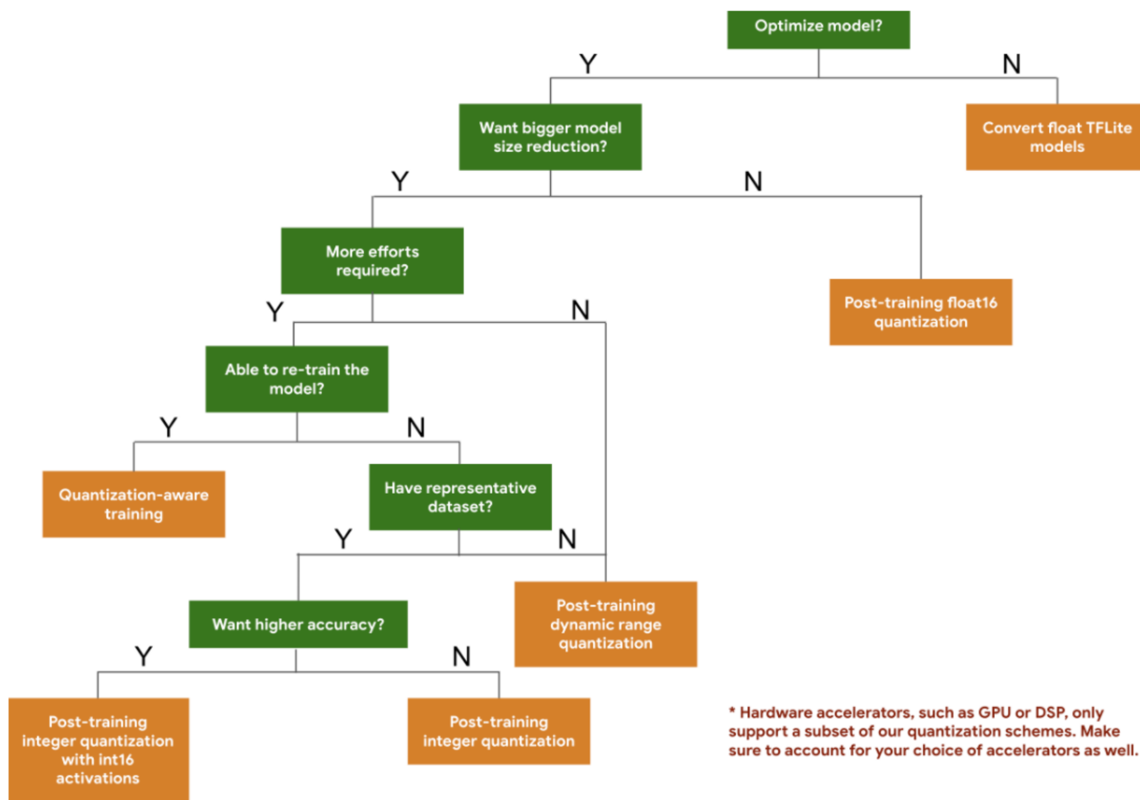


Figure 2-4. Quantization technique decision tree. [26]

Figure 2-4 shows a decision tree to help select the most adequate quantization method. It takes into account the size and the expected precision of the model.

2.3.2 Risks of quantization

NN are immensely complex functions, and in spite of being continuous, they can transform fastly. Quantization implies surrender accuracy. Therefore, for tasks in which safety is vital, it is precise to be excessively cautious. [26]

Capítulo 3 - Hardware devices and architectures used

3.1 Laptop i7 NVIDIA® GeForce® GTX

I used my Windows laptop to locally install DeepLab, as well as the GPU version of TensorFlow. Then I trained, evaluated and visualized the semantic segmentation of images using different models.

3.2 Virtual Machines

I configured a virtual machine to be able to execute DeepLab models. I first used VMware Workstation 15 player, nevertheless I had problems trying to get images from the webcam that I used. Because of that, I finally decided to employ VM VirtualBox, where I used Ubuntu 20.04 as the Operating System.

Overall, keeping the Virtual Machine (VM) in order was complicated because of all the continuous problems I encountered. As a consequence, I had to configure the VM multiple times (every time one got broken). I wrote a configuration script with all the necessary steps to correctly configure the VM, which can be found at Apéndice A.

3.3 Raspberry Pi 4 Model B 8GB RAM

Raspberry Pi 4 Model B is a small computer. In every new model there is a continuous improvement of the processor speed, multimedia performance, memory, and connectivity, while trying to keep similar power consumption. It brings desktop performance analogous to entry-level x86 PC systems. [34]

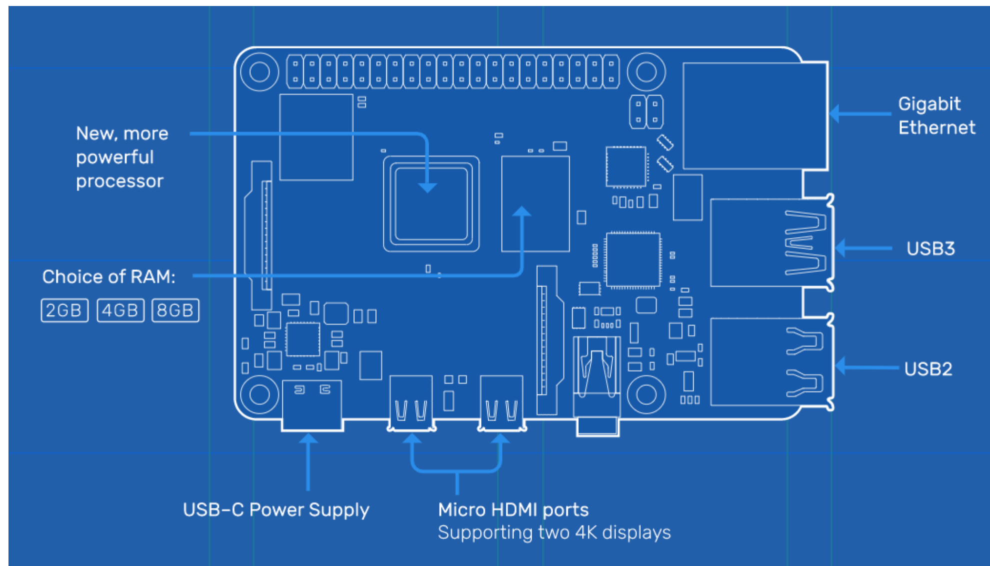


Figure 3-1. Raspberry Pi 4 Model B architecture. [33]

Between the key features, it offers a high-performance 64-bit quad-core processor, a pair of micro-HDMI ports which support 4K dual-display, dual-band 2.4/5.0 GHz wireless LAN, Bluetooth 5.0, Gigabit Ethernet, USB 3.0, and PoE capability [34] (Power over Ethernet, which grants the capability of supplying power to a networking device through the cable which also transmit the data). [48] In regards to graphics, it includes hardware video decode at up to 4Kp60, and video rendering to 1080p and 30fps and up to 8GB of RAM.

Finally, it must be emphasized that Raspberry Pi 4 Model B has 8GB of RAM LPDDR4. This is a random-access synchronous dynamic memory which is deployed in mobile devices due to its low energy consumption.

3.4 USB Accelerator Google Coral

"The Coral USB Accelerator is a USB device designed by Google that accelerates inference time for machine learning models. It provides an Edge TPU as a coprocessor for the computer" [46] (in my case, for the Raspberry Pi 4 Model B). It works for Linux, Mac, and Windows.

Figure 3-2. USB Accelerator Google Coral. [46]

To get started, it is necessary to download the Edge TPU runtime and the PyCoral library on the computer. [17] Then, I will use the accelerator to run TensorFlow Lite models, which are compatible with Edge TPU.

To be able to train my own models, I need to use post-training quantization, which will be explained later in this work.

3.4.1 Edge TPU

According to Google [8] and Coral [49] documentation, "Tensor Processing Units (TPUs) are Google's application-specific integrated circuits (ASICs)." "The *Edge TPU* provides high performance machine learning inferencing for low-power devices." "One Edge TPU can implement 4 trillion fixed-point operations per second (4TOPS) using 2 watts of power."

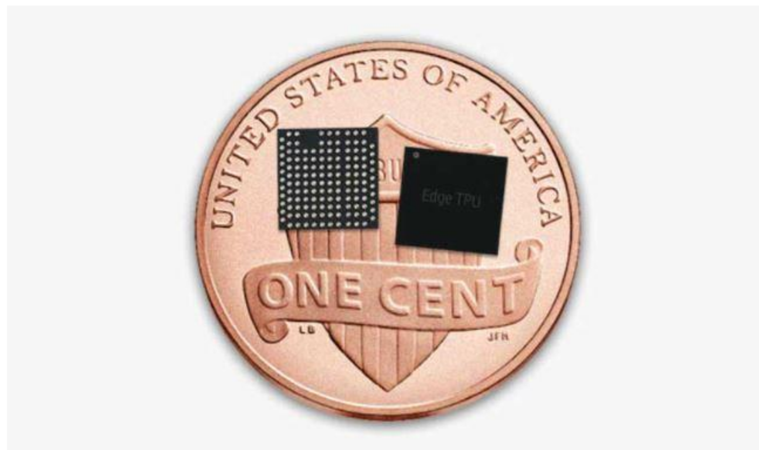


Figure 3-3. Two Edge TPU chips on a US penny. [49]

Edge TPU is an alternative to *Cloud TPUs*, also by Google. "Cloud TPUs run in a Google data center [...], and can perform 420 teraflops." "This means that they are perfect for training great, complex models. On the other hand, Edge TPU is ideal for small, low-power devices, since it provides extremely fast and power-efficient on-device ML inferencing." [49] It needs to be noted that Edge TPU only supports the TensorFlow Lite framework.

Capítulo 4 - Frameworks used

4.1 TensorFlow

TensorFlow is an open source platform for Machine Learning developed by the *Google Brain* team. "It offers its own ecosystem of tools, libraries and resources to enable the development of new innovations in the field of Machine Learning." [40]

4.2 TensorFlow Lite

Tensorflow Lite (TFLite) [42] is a set of tools that enables edge computing. It helps to run machine learning models on devices.

Its key features include:

- *Optimized for on-device machine learning*. It addresses latency (it is not cloud, so there is no round-trip to a server), it ensures privacy (no personal data spreads out of the device), there is no need for internet connectivity, there is reduced model and binary size and regarding power consumption, it offers efficient inference without network connections.
- *Multiple platform support*
- *Multiple language support*, such as Java, C++ and Python.
- *High performance*, with hardware acceleration and optimization of the models.
- *End-to-end examples* for frequent ML tasks.

Previously, some quantization techniques available for TFLite were discussed. Now, the process of how to create a TFLite model for the Edge TPU will be studied.

As a brief introduction, it can be stated that the main steps are:

1. *Picking* a new *model* or retraining one
2. *Converting* a TF (.pb) model into a compressed flat buffer (.tflite) with the TFLite Converter.
3. *Deploying*: taking the compressed .tflite file and loading it into a device.
4. *Optimizing*: quantizing the model.

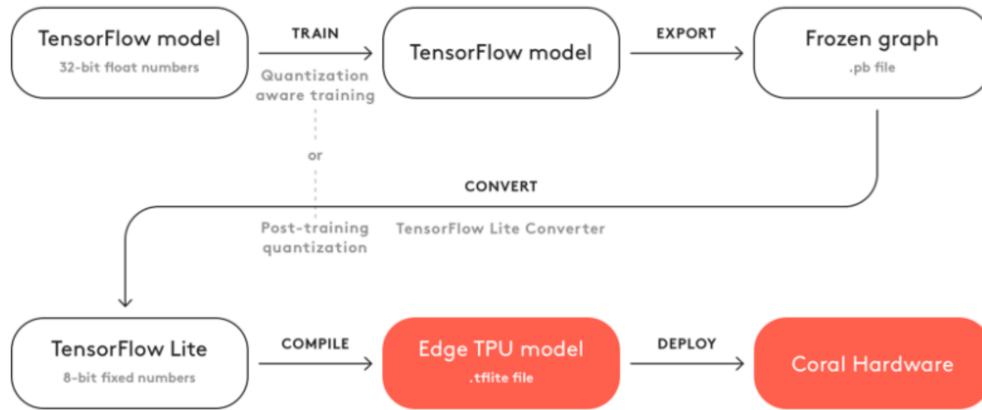


Figure 4-1. Workflow to create a model for Edge TPU. [13]

Figure 4-1 explains the two ways of generating the compatible Edge TPU model: *quantization-aware training*, and post-training quantization. These were previously introduced. As shown in the figure, *quantization-aware training* is a much broader process. For the sake of simplicity, I will focus on *post-training quantization.* [13]

4.2.1 Post-training quantization

"*Post-training quantization* [31] is a conversion technique that is able to reduce the model size as well as improving the [...] latency", without compromising the accuracy too much. The quantization can be implemented by converting a trained TensorFlow model to TFLite format using the TFLite Converter. As mentioned before, it is necessary to simplify models to be able to bring pretrained models to architectures with less resources.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Table 4-1. Post-training quantization techniques. [31]

4.2.1.1 Full integer post-training quantization

As *Table 4-1* indicates, there are several techniques to choose from. Given that Coral USB Accelerator will be used, the chosen technique is *Full integer quantization*, since it supports Edge TPU. Its main benefits are that it makes the model 4x smaller and provides 3x more speedup. [31]

This quantization technique is used to transform an already trained network into a quantized model, hence it does not require modifications to the network. [32]

"In addition, it offers further latency improvements, cuts down peak memory usage, and is compatible with integer-only hardware devices or accelerators" [31] (Coral in this case).

It is required to calibrate the range of all floating-point tensors in the model. To calibrate variable tensors like model input or outputs, it is necessary to run a few inference cycles with a small representative dataset. [16] Further in this project, I explained that for my own quantization I used dummy images and webcam images as representative datasets,

4.2.1.2 TensorFlow Lite converter

This converter transforms a TF model into a TFLite model (an optimized FlatBuffer *.tflite*). There are two ways to use the converter: by the *Python API* or by the *command line*.

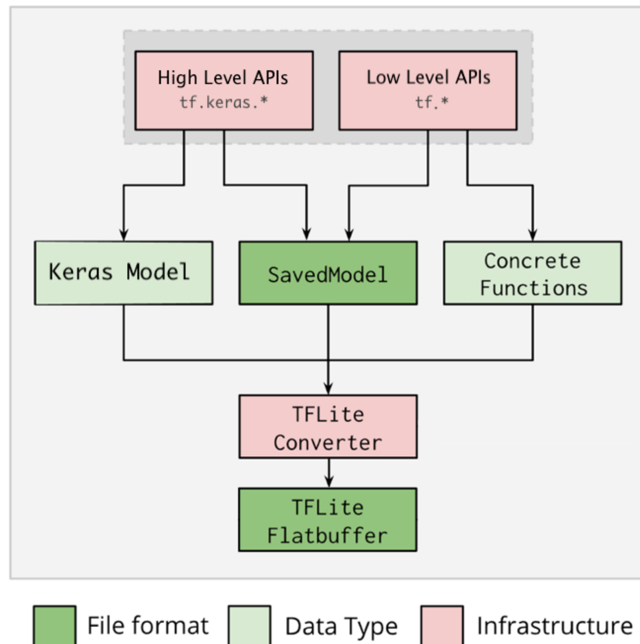


Figure 4-2. TFLite conversion diagram. [41]

To create a TFLite model for the Edge TPU, the first step is to convert the model to TFLite. Beware that to create a compatible model with *post-training quantization,* TensorFlow 1.15 must be the installed version (to set input and output type to *int8)*; it is not possible to use TensorFlow 2.0 because it only supports float inputs and outputs. Finally, it is necessary to compile the model for compatibility with the Edge TPU. [19]

4.3 DeepLab

"DeepLab [35] is a state-of-art deep learning model for semantic segmentation of images that aims to assign meaningful labels to all pixels of the input image." It was designed and open-sourced by Google in 2016. It has several features and many improvements have been made since then, counting DeepLab V2 [4], DeepLab V3 [5] and DeepLab V3+. [6]

Figure 4-3. Segmentation result example on Flickr image. [53]

DeepLab allows the users to train the model, evaluate the results in terms of mIoU (mean intersection-over-union) and visualize the segmentation results.

4.3.1 How does DeepLab work

To understand DeepLab we will focus on *DeepLabv3+* [6], its latest version. The model consists of two steps: [28]

- **Encoding phase**: The goal is to get the key information from the image. This will be done by a pre-trained CNN.
- **Decoding phase**: The previously extracted information is used to rebuild output of correct dimensions.

4.3.1.1 Spatial pyramid pooling

To guarantee that the model is robust to changes in object sizes, *spatial pyramid pooling (SPP)* networks are employed, which take multi-scale information because they use different scaled variants of the input during training. Then, the crucial features that can represent most information are combined.

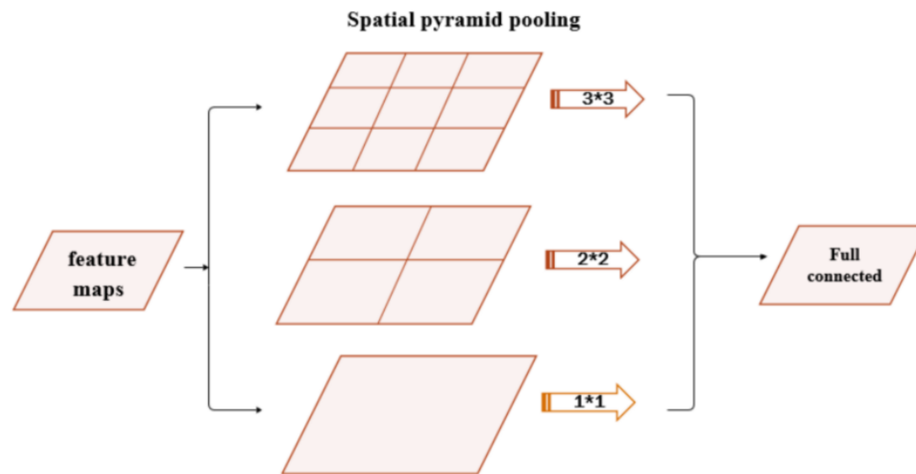


Figure 4-4. Spatial pyramid pooling. [28]

"Encoder-Decoder networks transform the input into a dense form that can represent all the input information." [28]

4.3.1.2 Atrous Convolutions

- To mitigate the enhancement in the computational and memory requirements of training caused by *SPP*, *autrous convolutions* are introduced.

- Atrous Convolutions get information from a broader effective field of view, but they keep equal computational complexity. [52] Its generalized form (in which normal convolution has ratio *r* =1) is:

$$y[i] = \sum_{k} x[i + r \cdot k] w[k]$$

Astrous Spatial Pyramid Pooling (ASPP) is the combination of *SPP* with *autrous convolutions.* They normally consist of 4 parallel operations.

4.3.1.3 Depthwise Separable Convolutions

This technique allows to reduce the computation number when performing convolutions. For instance, the input is 12 x 12 x 3 and a convolution of 5 x 5 is desired, which gives an output 8 x 8 x 1. For this purpose, the convolution is divided in two steps:

- *Depthwise convolution:* In this step the convolution 5 x 5 x 1 is performed, and the obtained output is 8 x 8 x 3.
- *Pointwise convolution:* Then, the channel number must increase. 1 x 1 kernels are used with 3 as the depth of the input. With this 1 x 1 x 3 convolution, the output is 8 x 8 x 1. To increase the number of channels, the convolutions 1 x 1 x 3 can be applied as much as wanted.

4.3.2 Datasets

I have studied mainly these these datasets to train the models, which have different characteristics:

- *PASCAL VOC 2012* [30]: It includes about 1400 images for training and validation and 20 object categories such as animals, vehicles and other daily life objects.
- *ADE20k [2]*: This dataset contains more than 27K images and over 3K object categories. It is very dense, as there are many objects in the images; this does not happen, for example, in PASCAL VOC 2012, which includes few objects in each image.
- *Cityscapes [7]*: It is focused on stereo video sequences recorded in streets and roads from 50 cities. There are about 3000 images classified in 30 categories and divided in 8 groups (nature, sky, humans…). These images have been chosen from key frames of the videos.

4.3.3 Model Zoo

This model zoo [51] offers DeepLab models trained on PASCAL VOC 2012, Cityscapes and ADE20K.

I have work the most with PASCAL VOC 2012, whose directory includes:

- A frozen inference graph: `frozen_inference_graph.pb`
- A checkpoint: `model.ckpt.data-00000-of-00001`, `model.ckpt.index`

These checkpoints used were:

- *mobilenetv2_coco_voc_trainaug*
- *mobilenetv2_coco_voc_trainval*
- *xception65_coco_voc_trainaug*
- *xception65_coco_voc_trainval*

The checkpoints have been pre-trained on the PASCAL VOC 2012 train_aug set or train_aug + trainval set.

Become aware that *MobileNet-v2* based models do not use ASPP (seen before) and decoder modules for fast computation. This implies that at a glance we could deduce that its accuracy is lower than in *Xception_68*.

Checkpoint name	Eval OS	Eval scales	Left-right Flip	Multiply-Adds	Runtime (sec)	PASCAL mIOU	File Size
mobilenetv2_coco_voc_trainaug	16 8	[1.0] [0.5:0.25:1.75]	No Yes	2.75B 152.59B	0.1 26.9	75.32% (val) 77.33 (val)	23MB
mobilenetv2_coco_voc_trainval	8	[0.5:0.25:1.75]	Yes	152.59B	26.9	80.25% (**test**)	23MB
xception65_coco_voc_trainaug	16 8	[1.0] [0.5:0.25:1.75]	No Yes	54.17B 3055.35B	0.7 223.2	82.20% (val) 83.58% (val)	439MB
xception65_coco_voc_trainval	8	[0.5:0.25:1.75]	Yes	3055.35B	223.2	87.80% (**test**)	439MB

Table 4-2. Computation complexity (in terms of Multiply-Adds and CPU Runtime) and segmentation performance (in terms of mIOU) on the PASCAL VOC val or test set. [51]

Looking at the PASCAL mIOU [10] it shows that the precision is indeed lower in *MobileNet-v2* than in *Xception_68*. However, the times are faster in *MobileNet-v2*.

Let's try the model with the lowest mIOU *(mobilenetv2_coco_voc_trainaug)* and the one with the highest mIOU *(xception65_coco_voc_trainval)* with this input *[1]*.

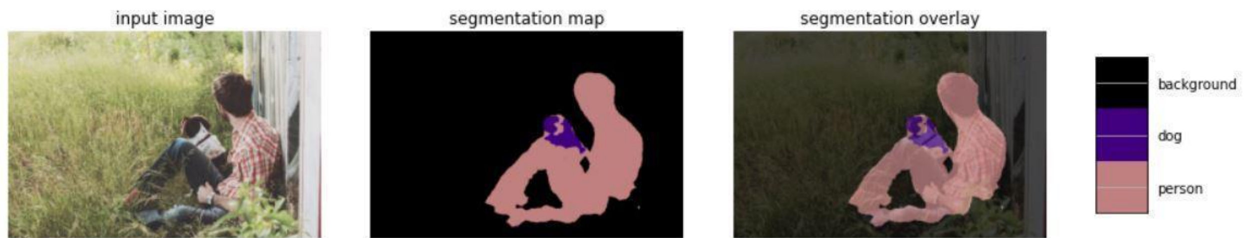


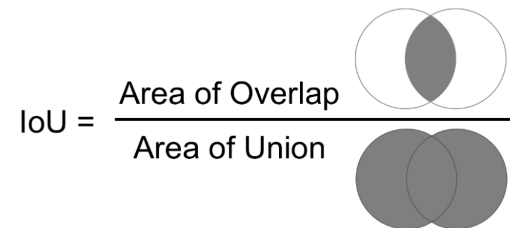
Figure 4-5. Example mobilenetv2_coco_voc_trainaug.

Figure 4-6. Example xception65_coco_voc_trainval

As said before, at a glance it is possible to see that its accuracy is higher in *Xception_68.*

4.3.4 *Evaluation metrics*

The metric in which the accuracy is measured is MIoU (Mean Intersection over Union).



It consists in calculating the IoU between the ground truth and the output predicted by the NN. This considers the region common to both and calculated the similarity percentage in comparison to the actual one. [10]

4.3.5 *Advantages of DeepLab*

There are three main advantages of using DeepLab: [4]

- *Speed*: thanks to atrous convolution.
- *Accuracy*: state-of-the-art results are obtained on complex datasets, such as PASCAL VOC 2012.
- *Simplicity*: this system consists of two fixed modules, DCNNs and CRFs.

4.4 OpenCV

OpenCV [27] is an Open source Computer Vision library. I have used it in the Python program that I wrote to be able to capture movement and recognize objects through my webcam.

Capítulo 5 - Implementation and experimentation

As seen in the project plan, after researching semantic segmentation and DeepLab, and experimenting with a DeepLab demo, I proceeded to install DeepLab and test models locally, both in my laptop and in a virtual machine.

5.1 DeepLab local installation

5.1.1 Laptop installation

As a first step, I cloned the git repository of DeepLap [35] in my Windows laptop with the help of Git Bash in order to implement a local installation. I installed a Tensorflow version compatible with GPU (*python -m pip install tensorflow-gpu==1.15.3*) as well as some libraries and drivers required for its correct functioning in GPU, such as *CUDA* 9. The software dependencies table can be found here [9].

When everything is installed, try the model_test.py to test that it is working correctly.

```
[       OK ] DeeplabModelTest.testForwardpassDeepLabv3plus
[ RUN      ] DeeplabModelTest.testWrongDeepLabVariant
[       OK ] DeeplabModelTest.testWrongDeepLabVariant
[ RUN      ] DeeplabModelTest.test_session
[  SKIPPED ] DeeplabModelTest.test_session
----------------------------------------------------------------------
Ran 5 tests in 25.231s

OK (skipped=1)
```

Figure 5-1. Console output when running model_test.py

5.1.1.1 Training

As a first step, I decided to start running DeepLab on the PASCAL VOC 2012 dataset. [30] The converted dataset will be saved at the directory ./deeplab/datasets/pascal_voc_seg/tfrecord following this structure:

```
+ datasets
  + pascal_voc_seg
    + VOCdevkit
      + VOC2012
        + JPEGImages
        + SegmentationClass
    + tfrecord
    + exp
      + train_on_train_set
        + train
        + eval
        + vis
```

Figure 5-2. Recommended Directory structure for Training and Evaluation [50]

Then, I trained the models using the train.py file (in the models/research/deeplab folder). The Flags can be modified according to the preferred requirements.

In this case, a training job using *xception_65* is done using this command:

```
# From tensorflow/models/research/
python deeplab/train.py \
    --logtostderr \
    --training_number_of_steps=30000 \
    --train_split="train" \
    --model_variant="xception_65" \
    --atrous_rates=6 \
    --atrous_rates=12 \
    --atrous_rates=18 \
    --output_stride=16 \
    --decoder_output_stride=4 \
    --train_crop_size="513,513" \
    --train_batch_size=1 \
    --dataset="pascal_voc_seg" \
    --tf_initial_checkpoint=${PATH_TO_INITIAL_CHECKPOINT} \
    --train_logdir=${PATH_TO_TRAIN_DIR} \
    --dataset_dir=${PATH_TO_DATASET}
```

Figure 5-3. Command used to run train.py. [50]

According to the DeepLab repository [50] , ${PATH_TO_INITIAL_CHECKPOINT} is the path to the initial checkpoint, ${PATH_TO_TRAIN_DIR} is the directory in which training checkpoints and events will be written to, and ${PATH_TO_DATASET} is the directory in which the PASCAL VOC 2012 dataset resides.

I specified the paths as follows:

- PATH_TO_INITIAL_CHECKPOINT=\deeplab\datasets\pascal_voc_seg\deeplabv3_pascal_trainval\model.ckpt
- PATH_TO_TRAIN_DIR=\deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\train
- PATH_TO_DATASET=\deeplab\datasets\pascal_voc_seg\tfrecord

```
INFO:tensorflow:global step 360: loss = 0.6684 (9.175 sec/step)
I1009 12:26:37.152738 19144 learning.py:507] global step 360: loss = 0.6684 (9.175 sec/step)
INFO:tensorflow:global step 370: loss = 3.5417 (9.094 sec/step)
I1009 12:28:08.706170 19144 learning.py:507] global step 370: loss = 3.5417 (9.094 sec/step)
INFO:tensorflow:global step 380: loss = 3.6274 (8.991 sec/step)
I1009 12:29:39.754368 19144 learning.py:507] global step 380: loss = 3.6274 (8.991 sec/step)
INFO:tensorflow:global step 390: loss = 0.8750 (9.085 sec/step)
I1009 12:31:14.799587 19144 learning.py:507] global step 390: loss = 0.8750 (9.085 sec/step)
INFO:tensorflow:global step 400: loss = 3.8645 (9.251 sec/step)
I1009 12:32:45.720869 19144 learning.py:507] global step 400: loss = 3.8645 (9.251 sec/step)
INFO:tensorflow:global step 410: loss = 0.2094 (9.298 sec/step)
I1009 12:34:19.366273 19144 learning.py:507] global step 410: loss = 0.2094 (9.298 sec/step)
INFO:tensorflow:global step 420: loss = 0.2929 (9.361 sec/step)
I1009 12:35:52.421364 19144 learning.py:507] global step 420: loss = 0.2929 (9.361 sec/step)
INFO:tensorflow:global_step/sec: 0.108336
I1009 12:35:55.778741 11168 supervisor.py:1099] global_step/sec: 0.108336
INFO:tensorflow:Recording summary at step 420.
I1009 12:36:00.562177 19544 supervisor.py:1050] Recording summary at step 420.
```

Figure 5-4. Terminal output when training DeepLab using the official repository (train.py)

This will train the model on the dataset and save the checkpoint files to *train_logdir*, which will later be used in the evaluation.

5.1.1.2 Evaluation

The official DeepLab repository offers an evaluation file (eval.py). To evaluate a model trained with *xception_65* this command is used:

```
# From tensorflow/models/research/
python deeplab/eval.py \
    --logtostderr \
    --eval_split="val" \
    --model_variant="xception_65" \
    --atrous_rates=6 \
    --atrous_rates=12 \
    --atrous_rates=18 \
    --output_stride=16 \
    --decoder_output_stride=4 \
    --eval_crop_size="513,513" \
    --dataset="pascal_voc_seg" \
    --checkpoint_dir=${PATH_TO_CHECKPOINT} \
    --eval_logdir=${PATH_TO_EVAL_DIR} \
    --dataset_dir=${PATH_TO_DATASET}
```

Figure 5-5. Command used to run eval.py. [50]

According to the DeepLab repository, ${PATH_TO_CHECKPOINT} is the path to the trained checkpoint (i.e., the path to train_logdir) and ${PATH_TO_EVAL_DIR} is the directory in which evaluation events will be written to.

I specified the paths as follows:

- PATH_TO_CHECKPOINT=\deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\train
- PATH_TO_EVAL_DIR=\deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\eval
- PATH_TO_DATASET=\deeplab\datasets\pascal_voc_seg\tfrecord

However, I could not get any relevant information about the evaluation. When running the evaluation script I got "nan" as a value of MIoU (instead of a number), which means "*not a number*". Despite trying to obtain a meaningful result, I could not find the solution.

5.1.1.3 Visualization

To perform inference employing DeepLab official repository the file vis.py is used. This command is used for a visualization job using *xception_65:*

```
# From tensorflow/models/research/
python deeplab/vis.py \
    --logtostderr \
    --vis_split="val" \
    --model_variant="xception_65" \
    --atrous_rates=6 \
    --atrous_rates=12 \
    --atrous_rates=18 \
    --output_stride=16 \
    --decoder_output_stride=4 \
    --vis_crop_size="513,513" \
    --dataset="pascal_voc_seg" \
    --checkpoint_dir=${PATH_TO_CHECKPOINT} \
    --vis_logdir=${PATH_TO_VIS_DIR} \
    --dataset_dir=${PATH_TO_DATASET}
```

Figure 5-6. Command used to run vis.py [50]

I specified the paths as follows:

- PATH_TO_CHECKPOINT=\deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\train
- PATH_TO_VIS_DIR=\deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\vis
- PATH_TO_DATASET=\deeplab\datasets\pascal_voc_seg\tfrecord

As a result, at \deeplab\datasets\pascal_voc_seg\exp\train_on_train_set\vis the segmentation results can be found:

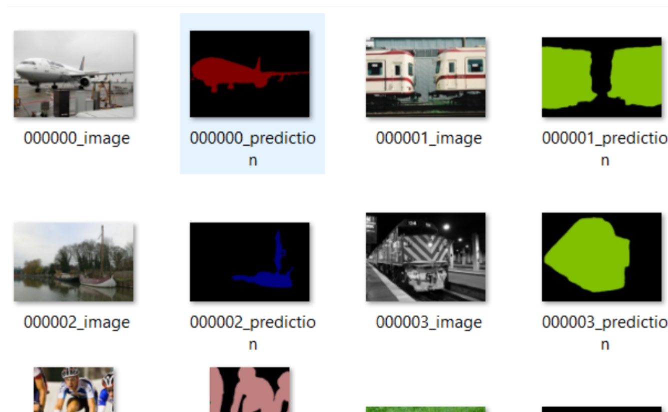


Figure 5-7. Segmentation results after running vis.py

5.1.1.4 Script (draft)

After experimenting with the local version, I created a script to implement semantic segmentation in given images and measure the execution times.

Firstly, I created a program to perform semantic segmentation on a sample input image; the outputs were semantic labels overlayed on the original image. This was based on a DeepLab Jupyter Demo. [12]

The program performed these tasks:

- Load the latest version of the pretrained DeepLab model
- Load the colormap from the PASCAL VOC dataset
- Assign colors to different labels (object types)
- Visualize the input image, and add the overlay of colors on the regions corresponding to different objects

I modified the program to measure inference execution times for different models as follows:

```
#for measuring time:
import time
start_time = time.time()
resized_im, seg_map = MODEL.run(original_im)
print("Execution time: %s seconds" % (time.time() - start_time))
```

Next, I tested the different models and collected their execution times. I measured each model 20 times to get the maximum and minimum values:

Model	Execution time
mobilenetv2_coco_voctrainaug	max: 0.937139 s min: 0.782747 s
mobilenetv2_coco_voctrainval	max: 1.179126 s min: 0.876957 s
xception_coco_voctrainaug	max: 9.069711 s min: 8.678269 s
xception_coco_voctrainval	max: 10.749623 s min: 8.941553 s

Table 5-1. Execution time (in seconds) of semantic segmentation using different models

In addition, I saved the output images as .png format. Here I present some examples of the results of the different models:

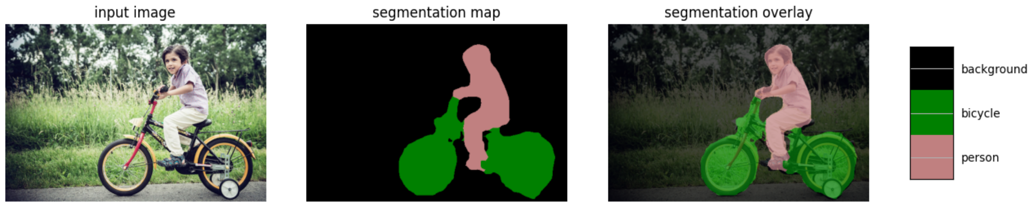


Figure 5-8. mobilenetv2_coco_voctrainaug segmentation laptop output example

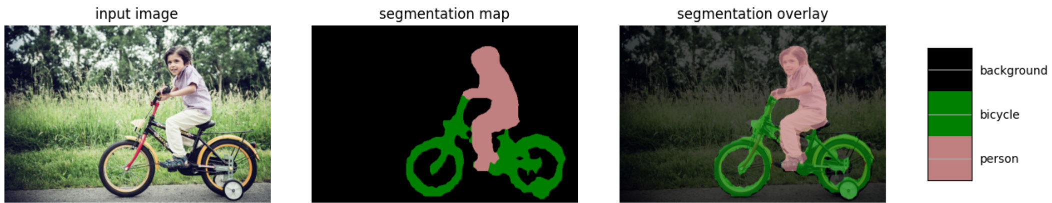


Figure 5-9. mobilenetv2_coco_voctrainval segmentation laptop output example

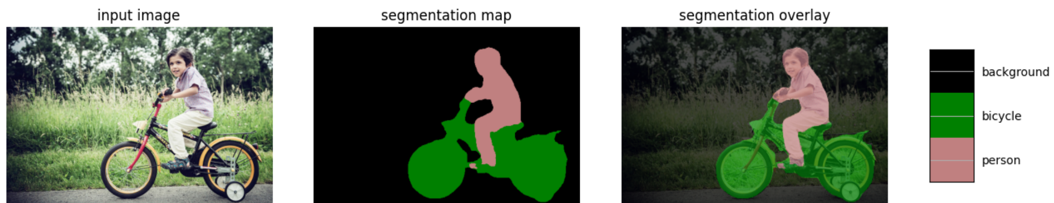


Figure 5-10. xception_coco_voctrainaug segmentation laptop output example

Figure 5-11. xception_coco_voctrainval segmentation laptop output example

As stated previously, *MobileNet-v2* models do not use ASPP and decoder modules for fast computation. *Table 5-1* shows that *-val* models have slightly higher inference execution times than its *-aug* version. It must be taken into consideration that the models have been pre-trained on PASCAL VOC 2012 train_aug set or train_aug + trainval set, which means that *-val* models have higher pretraining than *-aug* models. This could explain the higher accuracy of *xception_coco_voctrainval* and *mobilenetv2_coco_voctrainval* that can be observed in the figures above.

5.2 Virtual Machine

In the virtual machine I installed DeepLab locally just like I did in my laptop. However, after experimenting with the local version and obtaining more problems than relevant outcomes,

I will now present the execution times for different models which have now been run in the Virtual Machine with the earlier discussed script. I have used the exact same input image as before.

Model	Execution time
mobilenetv2_coco_voctrainaug	max: 1.37438035 s min: 1.037979 s
mobilenetv2_coco_voctrainval	max: 1.2677140 s min: 1.101976156 s
xception_coco_voctrainaug	max: 10.826671s min: 8.763001 s
xception_coco_voctrainval	max: 12.645364 s min: 9.4737105 s

Table 5-2. Execution time (in seconds) of semantic segmentation using different models in the Virtual Machine

Table 5-2 displays that the execution times in the Virtual Machine are slightly higher than the times of the models run directly in the laptop, as well as proportional. However, it seems that there is not much difference between the first two models. *mobilenetv2_coco_voctrainaug* has a higher maximum value than *mobilenetv2_coco_voctrainval*, unlike what happened in the laptop; but its minimum value is proportional to the results in the laptop.

I continued developing the script to implement semantic segmentation in webcam images and measuring different execution times. This new version of the script will be discussed later.

5.2.1 Virtual Machine configuration

The configuration of the virtual machine VM VirtualBox with all the settings necessary to run the script that I implemented to execute DeepLab models can be found at Apéndice A. I used Ubuntu 20.04 as the Operating System, and I activated a virtual environment with python 3.7 and installed TensorFlow 1.15.

5.2.2 Updated script

As an improvement of the program, with the help of the library OpenCV [27] the input image is captured through a webcam. Moreover, instead to compute the execution time as a whole, different times will be measured:

- Image capture time
- Image redimension time
- Inference time

5.2.3 *Measured times for each model*

Models	**Capture time**	**Inference time**	**Redimension time**
mobilenetv2_coco_voctrainaug	max_1: 0.744371 max2: 0.026295 min: 0.004427	max_1: 1.566632 max2: 1.184288 min: 1.079641	max: 0.042289 max2: 0.028708 min: 0.007805
mobilenetv2_coco_voctrainval	max_1: 0.712017 max2: 0.018788 min: 0.003908	max_1: 1.785160 max2: 1.590308 min: 1.144132	max: 0.033032 max2: 0.0235562 min: 0.012508
xception_coco_voctrainaug	max_1: 0.716405 max2: 0.013771 min: 0.004082	max_1: 11.369783 max2: 10.720958 min: 8.296429	max: 0.019892 max2: 0.015657 min: 0.005462
xception_coco_voctrainval	max_1: 0.742603 max2: 0.015859 min: 0.005048	max_1: 16.157057 max2: 9.322600 min: 8.121144	max: 0.025819 max2: 0.023149 min: 0.008753

Table 5-3. Semantic segmentation execution times (in seconds)of webcam images in the Virtual Machine

These are the maximums and minimum values of the capture, interface and redimensions times extracted from 20 iterations. Two maximums are presented because the first measurement has always the highest value (*max_1*) for capture and inference times. This first iteration is so high that it is not representative of the real times, because of that, *max2* is needed, which corresponds to a value independent from the first measurement. In redimension time this does not happen, but for the sake of coherence I have also included two *max* values.

Table 5-2 shows that the inference time is much higher for the *Xception* models, and, in particular, *xception_coco_voctrainaug* has the highest inference time range. Regarding Mobilenet v2, *mobilenetv2_coco_voctrainval* appears to have the highest times.

5.3 DeepLab in Raspberry Pi 4

As a further step, I used a Raspberry Pi as architecture to execute the program. To do so, I configured the device as explained in the next section. In addition, I captured the execution times as done before in the Virtual Machine.

5.3.1 *Raspberry Pi configuration*

Firstly, I installed Ubuntu 20.10 on a microSD, in order to use it as the Operating System of the Raspberry Pi. This process was very user friendly thanks to the installation interface. I followed this [20] guide.

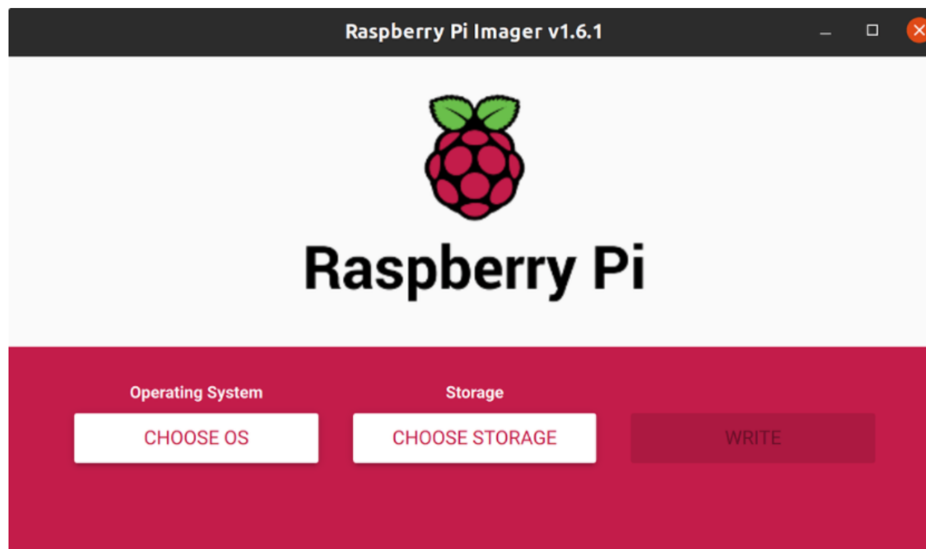


Figure 5-12. Raspberry Pi installation interface. [20]

I installed the Operating System successfully. However, I had certain issues regarding the software and libraries dependencies. I finally decided to install Ubuntu 20.04 on the microSD, which was definitely a more complex process. I installed Ubuntu Server 20.04 LTS on Raspberry Pi 4, as well as Ubuntu GNOME 3 desktop environment on it following this tutorial. [39]

Then I had to install all the software needed, for instance Tensorflow and OpenCV. In this case I installed *Tensorflow* 2.4 and the last *numpy* version. [21]

Finally, I connected the camera and executed my script, which ran successfully.

5.3.2 Measured times for each model

Models	**Capture time**	**Inference time**	**Redimension time**
mobilenetv2_coco_voctrainaug	max_1: 0.265220 max2: 0.003352 min: 0.001896	max_1: 3.361115 max2: 2.792313 min: 2.239232	max: 0.022485 max2: 0.022106 min: 0.019185
mobilenetv2_coco_voctrainval	max_1: 0.265913 max2: 0.003488 min: 0.0019512	max_1: 3.518164 max2: 2.715697 min: 2.364746	max: 0.023496 max2: 0.023123 min: 0.018469
xception_coco_voctrainaug	max_1: 0.261291 max2: 0.009885 max3: 0.003544 min: 0.001897	max_1: 33.316450 max2: 25.020431 min: 23.625263	max: 0.020824 max2: 0.020595 min: 0.018410
xception_coco_voctrainval	max_1: 0.263705 max2: 0.007603 max3: 0.002768 min: 0.002304	max_1: 32.254001 max2: 23.877033 min: 21.974183	max: 0.024083 max2: 0.020609 min: 0.018630

Table 5-4. Semantic segmentation execution times (in seconds) of webcam images in the Raspberry Pi 4

These results have also been extracted from 20 iterations. In this case, it happens the same as in the times measured in the Virtual Machine, which have been previously discussed. The first measurement has the highest values in Capture and Inference time, so we have *max_1* for that value, and *max2* for the next highest value. In some cases I have also included max3 because max2 was extremely high.

I can conclude that the semantic segmentation times executed in the Raspberry Pi are approximately twice as high as the results obtained in the Virtual Machine. Moreover, *xception_coco_voctrainaug* still has the highest values.

I will now show the segmentation results I captured through the webcam in different models and its characteristics.

Firstly, I used the model *xception_coco_voctrainval*; the one with the highest MIoU (87%), as seen in Table 4-2. I concluded that this model needs to have clear images as input, with good lightning and clear figures, to make a prediction. If the image is not totally clear, the output of the prediction is all "background". Moreover, when semantically segmenting an object, the borders are very accurately specified in the image, unlike in the models *xception_coco_voctrainaug* and *mobilenetv2_coco_voctrainaug,* which I measured later. It must be noted that when I performed the segmentation in both Xception models, the Raspberry Pi heated to a very high temperature and I had to refrigerate it. This happened due to the high inference time. I also concluded that *xception_coco_voctrainaug* is not worth the time it takes for the inference in relation to its accuracy (82-83% MIoU), which is almost the same as *mobilenetv2_coco_voctrainval* (80%) and has a very low inference time.

When using *mobilenetv2_coco_voctrainaug,* the model with the lowest MIoU, I noted that in spite of not having a very clear input image, this model makes a prediction. Sometimes the inference is correct, and sometimes it is not. Moreover, this model does not offer very accurate segmentation of the objects. The perimeter that it covers is generally larger than the actual object. However, the inference time is much lower in Mobilenet than in Xception. On the other hand, the accuracy is not much of a problem in *mobilenetv2_coco_voctrainval (*80% of MIoU versus 75 - 77% of *mobilenetv2_coco_voctrainaug)*, and it doesn not need totally clear images to make a right prediction.

I will now compare the inference behaviour of the models in similar situations.

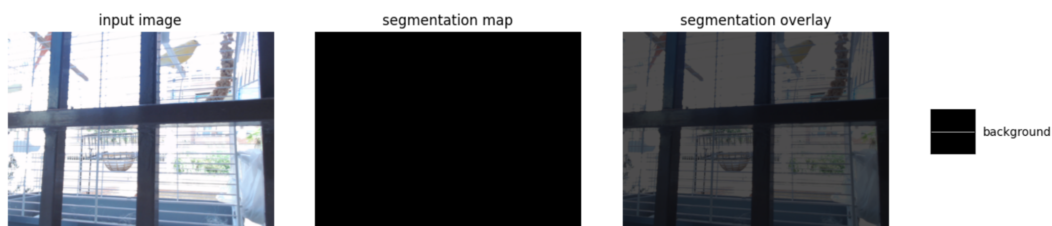


Figure 5-13. RasPi Unclear input, xception_coco_voctrainval 7

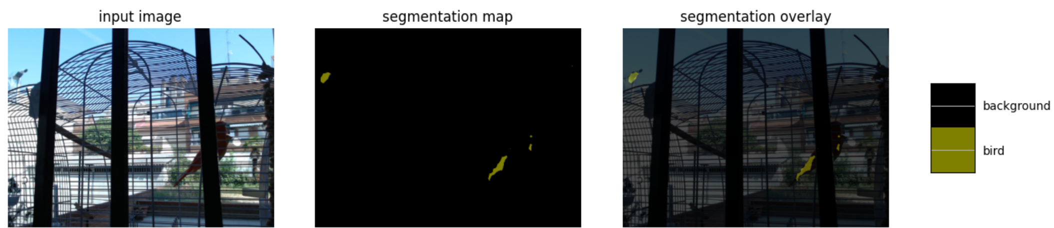


Figure 5-14. RasPi Unclear input, xception_coco_voctrainaug 26

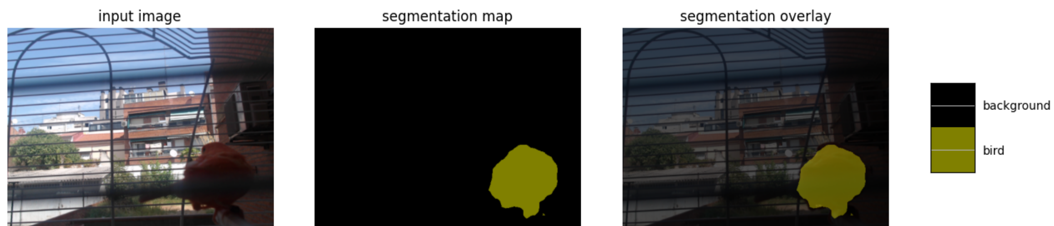


Figure 5-15. RasPi Unclear input, mobilenetv2_coco_voctrainval 29

Figure 5-16. RasPi Unclear input, alternative mobilenetv2_coco_voctrainval 31

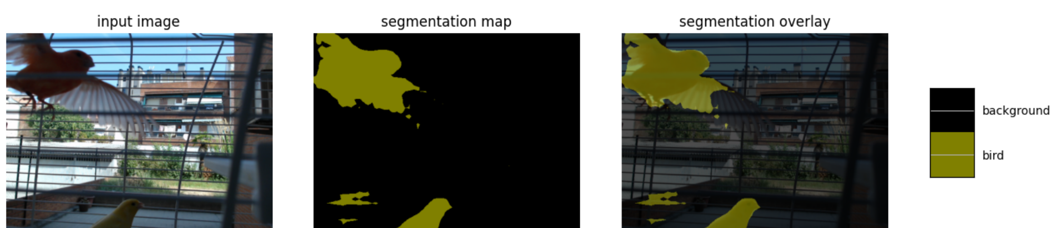


Figure 5-17. RasPi Unclear input, mobilenetv2_coco_voctrainaug 39

When the image is not totally clear (is too light or too dark), figures *5-13* to *5-17* show how the Xception models have more trouble segmenting the images when they do not know for sure which tag the object has. On the contrary, Mobilenet models

perform inference even when the image image is not clear. This can have right (*Figure 15*) or wrong (*Figure 16*) results.

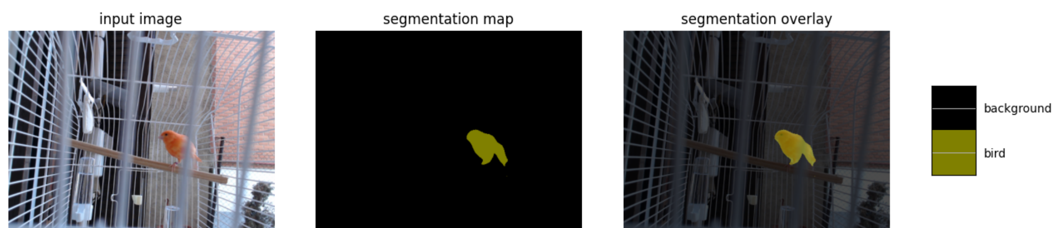


Figure 5-18. RasPi Clear input, xception_coco_voctrainval 9

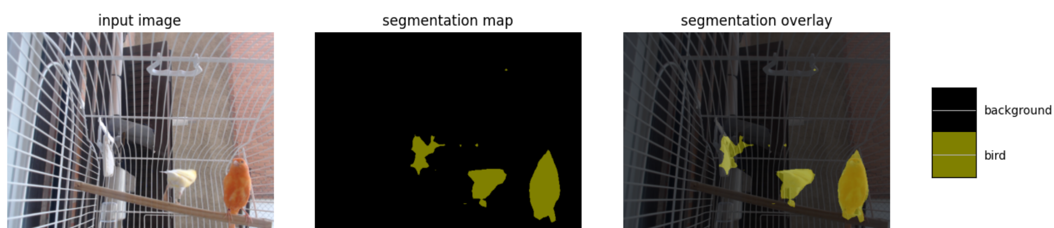


Figure 5-19. RasPi Clear input, xception_coco_voctrainaug

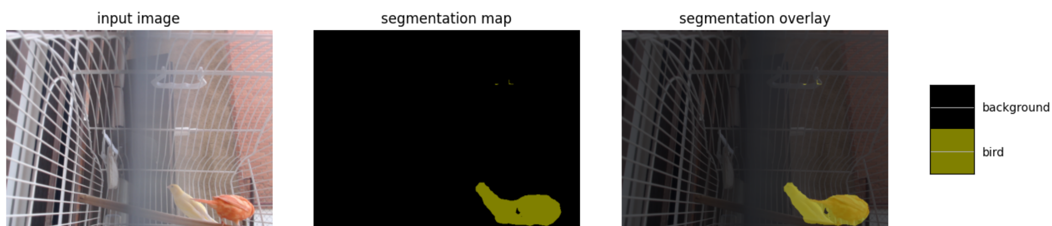


Figure 5-20. RasPi Clear input, mobilenetv2_coco_voctrainval 23

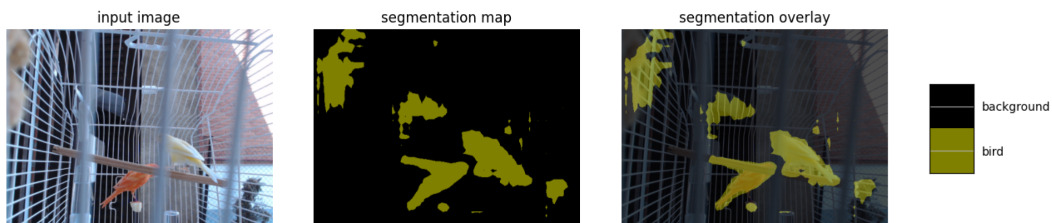


Figure 5-21. RasPi Clear input, mobilenetv2_coco_voctrainaug 30

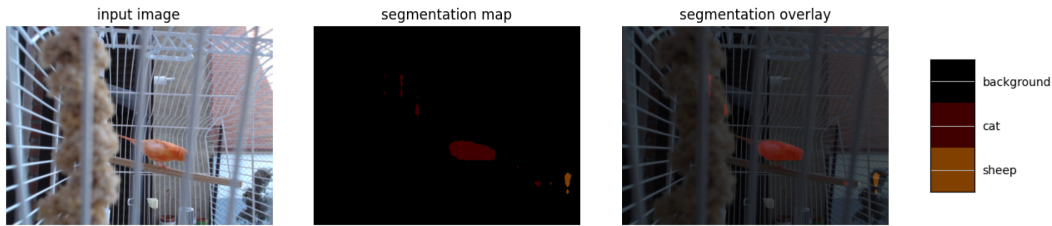


Figure 5-22. RasPi Clear input, alternative mobilenetv2_coco_voctrainaug 63

When the input image has good lighting, Xception models generally behave well, though *xception_coco_voctrainaug (Figure 19)* may not be as precise as *xception_coco_voctrainval (Figure 18)*. *mobilenetv2_coco_voctrainval* has correct behaviour too (*Figure 5-20*). However, *mobilenetv2_coco_voctrainaug* may identify the correct object, but highlight other areas in the image in which the object does not appear (*Figure 5-21*), or it can even indicate a different tag that does not correspond to the object in the input image (*Figure 5-22*).

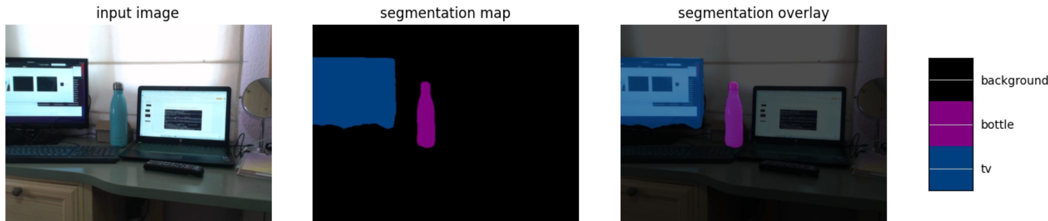


Figure 5-23. RasPi Several objects, xception_coco_voctrainval 47

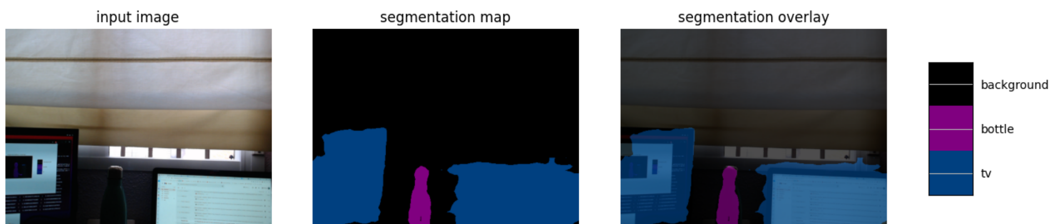


Figure 5-24. RasPi Several objects, xception_coco_voctrainaug

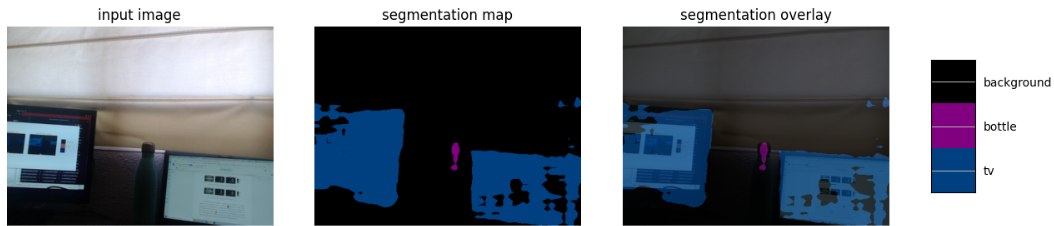


Figure 5-25. RasPi Several objects, mobilenetv2_coco_voctrainval 51

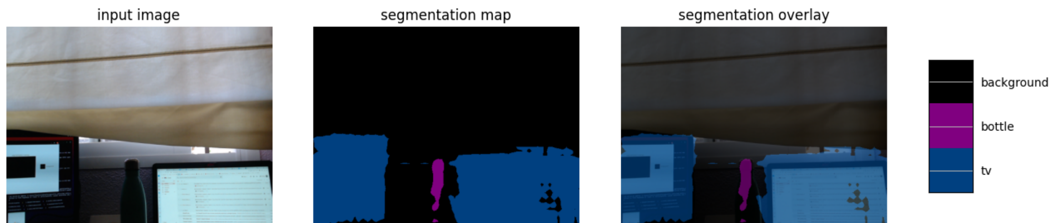


Figure 5-26. RasPi Several objects, mobilenetv2_coco_voctrainaug 9

In this situation there are various objects, and the lightning is not ideal. It is rare that in *xception_coco_voctrainval* the second monitor is not identified. However, the inference of the other objects is very accurate. The other thing to be noted is that, in this case, *mobilenetv2_coco_voctrainval* looks less accurate than *mobilenetv2_coco_voctrainaug*.

To conclude, using the model *mobilenetv2_coco_voctrainval* is a good solution because of its accuracy and its inference time. This model is not as accurate as *xception_coco_voctrainval*, but it is much faster (as it can be noted in the tables), which makes *mobilenetv2_coco_voctrainval* a good option if the precision is not extremely crucial.

5.4 Coral USB Accelerator in Raspberry Pi 4

To be able to bring pretrained models to architectures with less resources, it is necessary to simplify those models. This is done by *quantization*, a technique that creates models with smaller dimensions in TensorFlow Lite format.

5.4.1 Coral installation

To get started, it is necessary to download the Edge TPU runtime and the PyCoral library on the computer. All the installation steps are specified in the Coral documentation [17] for different Operating Systems.

5.4.2 How to run a model on the Edge TPU

Firstly, I plugged in the Coral USB. Then, I followed the Coral documentation to perform image classification with a MobileNet v2 using a Coral repository [22] to download the model, labels and a photo. The last step is to run the image classifier with the photo.

```
ubuntu@ubuntu:~/Desktop/coral/pycoral$ python3 examples/classify_image.py
\
> --model test_data/mobilenet_v2_1.0_224_inat_bird_quant_edgetpu.tflite \
> --labels test_data/inat_bird_labels.txt \
> --input test_data/parrot.jpg
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading
the model into Edge TPU memory.
18.0ms
4.7ms
4.8ms
5.5ms
5.4ms
-------RESULTS--------
Ara macao (Scarlet Macaw): 0.75781
```

Figure 5-27. Edge TPU inference results

Figure 5-27 shows the results of the performed inference on the Edge TPU using TensorFlow Lite. The top object classification label shows its confidence score, from 0 to 1.0. In this case, the program is 0.75781 sure that the image represents a scarlet macaw.

5.4.3 How to run TFLite object detection models on the Raspberry Pi

For this next procedure, I used this [15] repository following this guide [14]. I had already installed Tensorflow and OpenCV in my Raspberry Pi, so I saved those steps. To set up the TensorFlow Lite (TFLite) detection model, I used one Google's sample TFLite model, which can be found in the Sample_TFLite_model folder. The sample is a quantized *SSDLite-MobileNet-v2* object detection model which is trained off the MSCOCO dataset and converted to run on TFLite.

To run the TFLite model, I executed the following command:

```
python3 TFLite_detection_webcam.py --modeldir=Sample_TFLite_mode
```

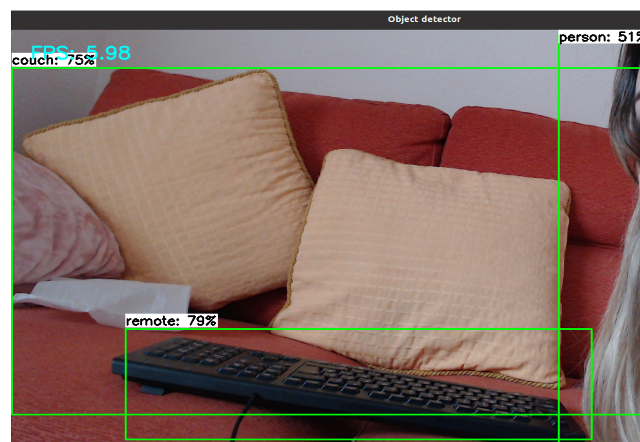


Figure 5-28. TFLite object detection models on a Raspberry Pi

Figure 5-28 shows that the webcam is used to capture the input video. The object detection takes place in real time. As the image shows, the model does not implement semantic segmentation, but just object detection. The difference is that in semantic segmentation, each pixel is labeled with an object class, unlike what happens in object detection.

At the top right corner, *Figure 5-28* displays "*5,98 FPS*". Frames Per Second determines how fast the object detection model processes the input video and creates the output. [38]

5.4.4 How to run Edge TPU object detection models on the Raspberry Pi using Coral USB Accelerator

As explained before, the Coral USB Accelerator uses the Edge TPU, which is an ASIC chip with highly parallelized ALUs (arithmetic logic units) that are directly connected to each other, unlike what happens in GPUs. [14]

All this makes object detection models run faster. To set up Coral the first step is to install the *libedgetpu1* library. There are two options, but I chose *libedgetpu1-std*, which allows 22.6 FPS (frames per second). The other option, *ibedgetpu1-max*, allows 26.1 FPS, but Coral would get hotter.

```
sudo apt-get install libedgetpu1-std
```

The next step is to set up the Edge TPU detection model. This model is compiled specifically to run on Edge TPU devices like Coral, and is kept in a *.tflite* file. There are two alternatives:

- Using Google's sample model, which is compiled from the quantized *SSDLite-MobileNet-v2* we used in the previous step.
- Using my own custom EdgeTPU model. However, *edgetpu-compiler* package does not work on the Raspberry Pi, but on a Linux PC.

Finally, I used Google's sample model to get a better comparison with the previous step.

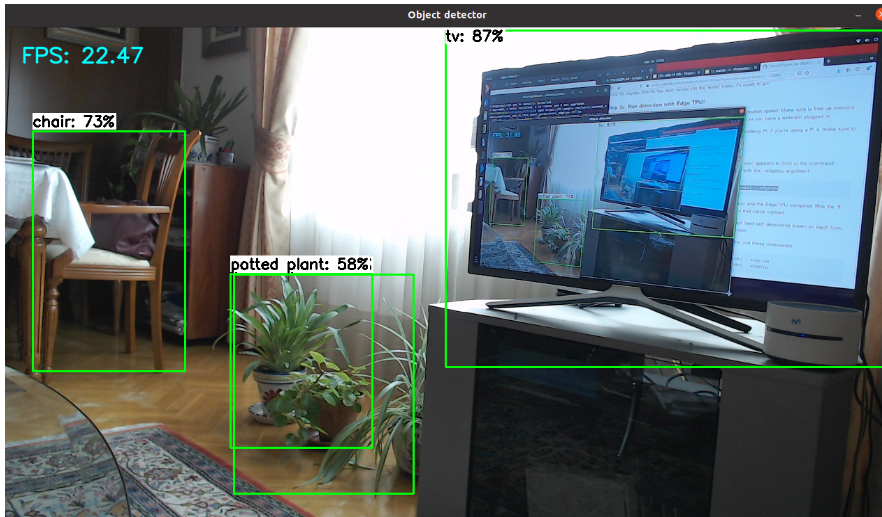


Figure 5-29. Edge TPU object detection models on a Raspberry Pi using Coral

Figure 5-29 shows the object detection. At the top right corner, it displays "*22,47 FPS*". In comparison to the TFLite object detection model on a Raspberry Pi without Coral (5,98 FPS), the usage of Coral USB Accelerator has obviously increased the speed at which the object detection model processes the video and generates the desired output. [38]

This means that the average inference performance (which is the same as frames-per-second) computed by the total inferences performed within the total of timing windows. [43]

5.4.5 How to quantize a model to run it on Edge TPU

As explained earlier, to create a model for Edge TPU runnable by the Coral USB Accelerator, the TensorFlow model needs to be quantized. I will focus on post-training quantization. The main steps are to use the TensorFlow Lite Converter to obtain a TF Lite model (.tflite), and then compile this model specifically for Edge TPU to get an Edge TPU model, which can then be deployed to the Coral hardware.

To use the TF Lite Converter the Python API or the command line can be used. All these conversion steps were done in the Virtual Machine, because it would not work to implement it on the Raspberry Pi.

In my case, I used the `tf.compat.v1.lite.TFLiteConverter` Python API to convert a frozen graph (.pb) from a file. I used the model Mobilenet_1.0_224, which is small, low-latency and low-power. For it, I created a script in which I first converted the model and then saved it as a *.tflite* file.

```
(envir2) tfgdeeplab@tfgdeeplab-VirtualBox:~/Desktop$ python3 convert_fg_lite_ne
w.py
2021-08-27 17:31:37.493777: I tensorflow/core/grappler/optimizers/meta_optimize
r.cc:788]   constant_folding: Graph size after: 174 nodes (0), 173 edges (0), t
ime = 26.891ms.
```

Figure 5-30. Terminal output when converting the TF model to a TF Lite model

Once I had the TFLite file, the next step was to compile it for Edge TPU, which I did following the Coral compiling documentation. [13] Firstly, I had to follow some instructions to install the compiler (`edgetpu-compiler`) on my Linux system. To use the compiler, the following command is used:

```
edgetpu_compiler [options] model…
```

```
(envir2) tfgdeeplab@tfgdeeplab-VirtualBox:~/Desktop$ edgetpu_compiler mobv2_1_0
_224.tflite
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 97 ms.

Input model: mobv2_1_0_224.tflite
Input size: 2.18MiB
Output model: mobv2_1_0_224_edgetpu.tflite
Output size: 2.18MiB
On-chip memory used for caching model parameters: 0.00B
On-chip memory remaining for caching model parameters: 0.00B
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 0
Total number of operations: 77
Operation log: mobv2_1_0_224_edgetpu.log

Model successfully compiled but not all operations are supported by the Edge TP
U. A percentage of the model will instead run on the CPU, which is slower. If p
ossible, consider updating your model to use only operations supported by the E
dge TPU. For details, visit g.co/coral/model-reqs.
Number of operations that will run on Edge TPU: 0
Number of operations that will run on CPU: 77
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

Figure 5-31. Terminal output when compiling the TF Lite model to Edge TPU

When the model has compiled successfully, it generates a *_edgetpu.tflite* file.

Then, it is time to run this model on the Raspberry Pi with PyCoral API. The first thing to do is to install PyCoral with this command:

```
sudo apt-get install python3-pycoral
```

After this, I tried an inferencing example from Coral to test that the model worked correctly.

```
ubuntu@ubuntu:~/Desktop/TFG$ python3 deeplab_esther_web_quanti.py
digital clock: 0.24517
```

Unfortunately, it was not the case. I tested a cat image and it detected a clock. In addition, this model was not adequate to perform semantic segmentation. I later noticed that *Figure 5-31* indicates that no operations would be performed in Edge TPU, which means that the type of quantization that I needed was not successful.

Then, I created a Google Colab program to perform the quantization, which behaved properly. I needed to use a representative dataset to perform the full integer post-training quantization and be able to run the resulting model in an Edge TPU architecture (the Coral Accelerator).

For this purpose, I first generated a dataset of *dummy* images, which are randomly generated, to test that it worked properly. I later created my own dataset of 30 images taken from my own webcam to use them as the representative dataset. Once this step was completed, the *mobilenetv2_dm05_coco_voc_trainval* model was transformed into a TF Lite model.

I compared the bytes saved with the quantization using the dummy images:

```
    TensorFlow Model is 3042785 bytes
TFLite Model is 963896 bytes
Post training int8 quantization saves 2078889 bytes
```

And my webcam images dataset:

```
TensorFlow Model is  3042785 bytes
TFLite Model is 2789284 bytes
Post training int8 quantization saves 253501 bytes
```

The higher the quality of the representative dataset, the smaller number of bytes saved in the quantization.

The next step is to compile the TF Lite model to Edge TPU (to get a *_edgetpu.tflite* model) and to download it to perform inference on the Raspberry Pi using the Coral Accelerator. It must be noted that the only model that I found whose operations can run for EdgeTPU is *mobilenetv2_dm05_coco_voc_trainval.*

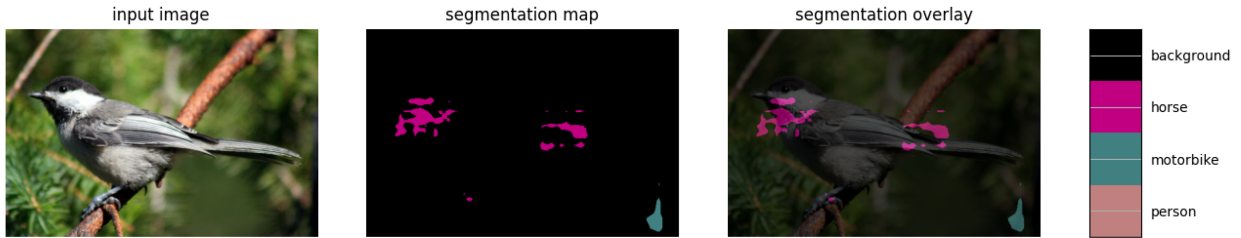


Figure 5-32. Inference result of dummy_edgetpu.tflite model with input image

When I used the *dummy_edgetpu.tflite* model (the model quantized using dummy images as a representative dataset) using a downloaded image as an input, I got the results displayed in *Figure 5-32*.

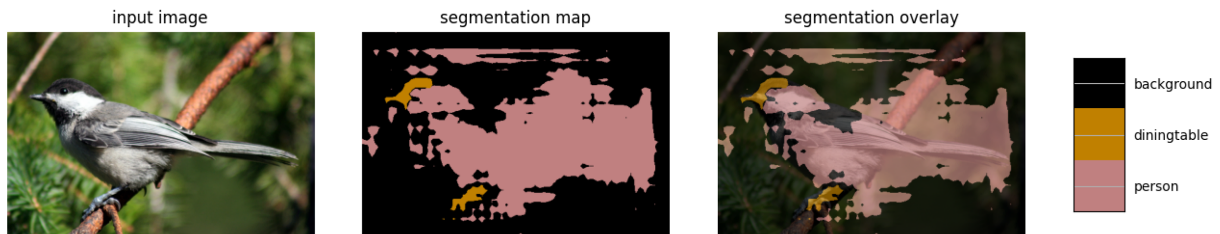


Figure 5-33. Inference result of smarty_edgetpu.tflite model with input image

And when I used the *smarty_edgetpu.tflite* model (the model quantized using webcam images as a representative dataset) using a downloaded image as an input, I got the result of *Figure 5-33*. It can be noted that neither are very accurate.

To perform inference with these models I created a script, which can be executed as follows:

```
python3 sem_seg_edgetpu_pics.py --model dummy_edgetpu.tflite
--input bird.bmp --keep_aspect_ratio --output
segmentation_result_dummy.jpg
```

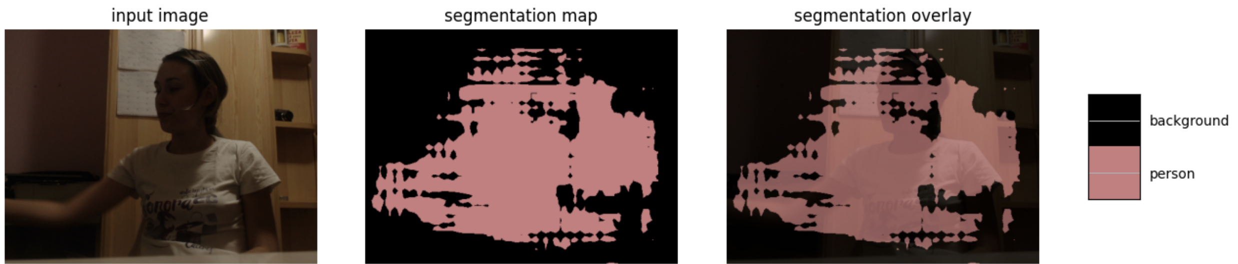


Figure 5-34. Inference result of smarty_edgetpu.tflite model with webcam input

However, when using the *dummy* model with a webcam image input, the inference results were almost always labeled as "background" even though there were several objects present in those images. Nevertheless, *smarty* models predicted some object labels even if they were not always correct or accurate, as we can see in this example in *Figure 5-34*.

Regarding inference time, the *dummy* model takes a little longer than the *smarty* model. A comparison between the EdgeTPU models and the non-quantized models will be presented later.

Capítulo 6 - Results Comparison and Analysis

The inference, capture and redimension times of the different models in various architectures have already been already shown. In this section the results will be compared and analyzed to comprehend better their meaning.

6.1 Laptop vs. Virtual Machine execution times

In the overall execution time results of the semantic segmentation executed in the laptop (*Table 5-1*) are slightly lower than the execution times in the Virtual Machine (*Table 5-2*). There is not a big difference between the models' behaviour depending on the architecture used; the times in the laptop are proportional to the times in the virtual machine. These results were computed with the same program using the same input images.

6.2 Virtual Machine image input vs. Virtual Machine webcam input

Later, I implemented some modifications in the program to measure the inference, capture and redimension times on their own, unlike the tables before where all these times were considered the execution time. This time the program did not perform the semantic segmentation on pictures given as input, but on images captured through the webcam, hence the capture time was also measured. (*Table 5-3*)

In this table it can be noted that the capture time has no significant variance between the models. Moreover, the first capture time of each model is significantly higher than the rest. This phenomenon happens during the inference as well, but not so much during the redimension time.

When comparing the inference of the webcam images with the execution time of the input images, it can be seen that the times are lower in the laptop than in the virtual machine for the *Mobilenet V2* models. All this taking into account that the execution time of the laptop includes the inference and redimension time. However, for the *Xception* models this is more ambiguous, but we can see that the minimum values are somewhat smaller in the virtual machine than in the laptop.

Finally, the redimension time has the lowest values for the *Xception* models, specifically the *xception_coco_voctrainaug* model; while the Mobilenet V2 models have higher redimension times.

6.3 Virtual Machine vs. Raspberry Pi

When it comes to the Raspberry Pi, the capture times are significantly smaller than in the Virtual Machine, and there is not much time difference between the models. (Table 5-4)

Models	**Virtual Machine inference time**	**Raspberry Pi inference time**
mobilenetv2_coco_voctrainaug	max_1: 1.566632 max2: 1.184288 min: 1.079641	max_1: 3.361115 max2: 2.792313 min: 2.239232
mobilenetv2_coco_voctrainval	max_1: 1.785160 max2: 1.590308 min: 1.144132	max_1: 3.518164 max2: 2.715697 min: 2.364746
xception_coco_voctrainaug	max_1: 11.369783 max2: 10.720958 min: 8.296429	max_1: 33.316450 max2: 25.020431 min: 23.625263
xception_coco_voctrainval	max_1: 16.157057 max2: 9.322600 min: 8.121144	max_1: 32.254001 max2: 23.877033 min: 21.974183

Table 6-1. Inference time comparison (in seconds) between Virtual Machine and Raspberry Pi

However, the inference time in the RasPi is approximately two times higher than in the Virtual Machine. *Table 6-1* shows the inference time comparison between both architectures.

Regarding the redimension time, the *Xception* models have smaller values in the Virtual Machine than in the Raspberry Pi. In particular, the minimum values are extremely small, while the maximum values are very similar to the ones in the RasPi.

6.4 Raspberry Pi: regular model vs. Edge TPU model

The quantization and EdgeTPU compilation were performed on a *mobilenetv2_dm05_coco_voc_trainval* model. This will be a comparison of the capture, between the non-quantized (regular) model and the *_edgetpu.tflite* model.

Model	**Capture time**	**Inference time**	**Redimension time**
regular	max1: 0.303826 max2: 0.0177173 min: 0.001894	max1: 5.633136 max2: 2.095738 min: 1.611738	max: 0.058004 min: 0.020867
dummy_edgetpu.tflite	max: 0.269128 min: 0.266331	max: 0.008964 min: 0.008219	max: 0.033319 min: 0.029464
smarty_edgetpu.tflite	max: 0.273280 min: 0.268579	max: 0.007211 min: 0.007089	max: 0.031751 min: 0.030280

Table 6-2. Time comparison (in seconds) between regular and EdgeTPU semantic segmentation model in Raspberry Pi

The table shows, as we have seen in previous regular models, that *mobilenetv2_dm05_coco_voc_trainval* regular model always has a higher capture and inference time on the first iteration (max1), but this pattern does not happen for the redimension time.

In the *_edgetpu.tflite* models I implemented a script which performs just one operation each time it is run, unlike the script of the regular model, which has a loop that captures webcam images until it is specified that it should stop. Because of this, the capture times of the *_edgetpu.tflite* models are higher (all the times the capture time is measured, it is considered the first (and only) time during that execution, and therefore the highest). Overall, it seems like the capture time of the regular model may be a little higher.

With regards to the inference, we see that in the *_edgetpu.tflite* models the time is extremely lower thanks to the Coral Accelerator. Even in the case that it happened the same as with the capture time, that means that the inference time could be even lower in the next iterations if there was a loop in the script.

On the other hand, it is interesting to see that the times for the *_edgetpu.tflite* models have very similar maximum and minimum values. The redimension time seems very similar for all models, but as said previously, the difference between maximum and minimum values is very small for *_edgetpu.tflite* models.

Moreover, the *dummy* model has slightly higher inference time than the *smarty* model.

Capítulo 7 - Conclusions and future work

Semantic segmentation has become essential nowadays. Even everyday people use this technology on their phones, and it should be possible for everyone to be able to take advantage of it. Thanks to the advances in Edge Computing, this is now possible.

This project shows that the results obtained are positive. By simplifying (quantization) a semantic segmentation model and using an USB Coral Accelerator I was able to perform inference in real-time on images captured through the webcam on a device restricted by computing power and energy consumption (Raspberry Pi).

Human-eye vision cannot differentiate a succession of images in 24 FPS, which is equivalent to 0.0416 seconds. This means that we perceive it as a video, not as independent images. I obtained an average inference time of 0.0085 and 0.0071 seconds in different EdgeTPU models, hence if the script which runs the inference had a loop, the semantic segmentation would operate like a video, in real time. However, to get these results, we have to sacrifice accuracy.

Nonetheless, using a regular (non-quantized) semantic segmentation model on the Raspberry Pi, or even a Linux virtual machine, and using DeepLab as a basis, an accurate implementation is possible although inference times are not efficient.

The next steps would be to deploy DeepLab to other platforms such as Intel NCS and Imagination Technologies PowerVR.

Bibliografía

1. (n.d.). picsum. https://i.picsum.photos/id/1012/3973/2639.jpg?hmac=s2eybz51lnKy2ZHkE2wsgc6S81fVD1W2NKYOSh8bzDc
2. *ADE20K*. (n.d.). MIT CSAIL. https://groups.csail.mit.edu/vision/datasets/ADE20K/
3. Briot, A., Viswanath, P., & Yogamani, S. (2018). *Analysis of Efficient CNN Design Techniques for Semantic Segmentation*. IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). 10.1109/CVPRW.2018.00109
4. Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2016). DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *40*, pp. 834-848. 10.1109/TPAMI.2017.2699184
5. Chen, L.-C., Papandreou, G., Schroff, F., & Adam, H. (2017). Rethinking Atrous Convolution for Semantic Image Segmentation. (arXiv:1706.05587).
6. Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., & Adam, H. (2018). Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. *European Conference on Computer Vision (ECCV)*, pp 833-851.
7. *The Cityscapes Dataset*. (n.d.). https://www.cityscapes-dataset.com
8. *Cloud Tensor Processing Units (TPUs)*. (n.d.). Google Cloud. https://cloud.google.com/tpu/docs/tpus
9. *Configuraciones de compilación probadas*. (n.d.). TensorFlow. https://www.tensorflow.org/install/source?hl=es-419#tested_build_configurations
10. CYBORG NITR. (2020, May 9). *MIoU Calculation*. Medium. https://medium.com/@cyborg.team.nitr/miou-calculation-4875f918f4cb

11. Danka, T. (2020, Jun 29). *How to accelerate and compress neural networks with quantization*. Towards Data Science. https://towardsdatascience.com/how-to-accelerate-and-compress-neural-networks-with-quantization-edfbbabb6af7

12. *DeepLab Demo*. (n.d.). https://colab.research.google.com/github/tensorflow/models/blob/master/research/deeplab/deeplab_demo.ipynb?authuser=1

13. *Edge TPU Compatibility overview*. (n.d.). Coral. https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

14. EdjeElectronics. (n.d.). *Part 2 - How to Run TensorFlow Lite Object Detection Models on the Raspberry Pi (with Optional Coral USB Accelerator)*. GitHub. https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi/blob/master/Raspberry_Pi_Guide.md

15. EdjeElectronics. (n.d.). *TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi*. GitHub. https://github.com/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi

16. *Full integer quantization*. (n.d.). TensorFlow. https://www.tensorflow.org/lite/performance/post_training_quantization#full_integer_quantization

17. *Get started with the USB Accelerator*. (n.d.). Coral. https://coral.ai/docs/accelerator/get-started

18. Hasan, T. (2018, May 12). *Semantic Segmentation*. Data Science Portfolio. https://tariq-hasan.github.io/concepts/computer-vision-semantic-segmentation/

19. *How do I create a TensorFlow Lite model for the Edge TPU?* (n.d.). Coral. https://coral.ai/docs/edgetpu/faq/#how-do-i-create-a-tensorflow-lite-model-for-the-edge-tpu
20. *How to install Ubuntu Desktop on Raspberry Pi 4*. (n.d.). Ubuntu. https://ubuntu.com/tutorials/how-to-install-ubuntu-desktop-on-raspberry-pi-4#1-overview
21. *Install Ubuntu 20.04 + OpenCV + TensorFlow (Lite) on Raspberry Pi 4*. (n.d.). Q-engineering. https://qengineering.eu/install-ubuntu-20.04-on-raspberry-pi-4.html
22. Kovalev, D. (n.d.). *PyCoral API*. GitHub. https://github.com/google-coral/pycoral
23. Makaya, C., Iyer, A., Salfity, J., Athreya, M., & Lewis, M. A. (2020). *Cost-effective Machine Learning Inference Offload for Edge Computing*. arXiv:2012.04063.
24. Marcelino, P. (2018, Oct 23). *Transfer learning from pre-trained models*. Towards Data Science. https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751
25. Mihajlovic, I. (2019, Apr 25). *Everything You Ever Wanted To Know About Computer Vision*. Towards Data Science. https://towardsdatascience.com/everything-you-ever-wanted-to-know-about-computer-vision-heres-a-look-why-it-s-so-awesome-e8a58dfb641e
26. *Model optimization*. (n.d.). TensorFlow. https://www.tensorflow.org/lite/performance/model_optimization
27. *OpenCV*. (n.d.). https://opencv.org
28. Pal, S. (2019, February 26). *Semantic Segmentation: Introduction to the Deep Learning Technique Behind Google Pixel's Camera!* Analytics Vidhya.

https://www.analyticsvidhya.com/blog/2019/02/tutorial-semantic-segmentation-google-deeplab/

29. Pan, S. J., & Yang, Q. (2009). A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(Issue 10). 10.1109/TKDE.2009.191
30. *PASCAL VOC 2012 dataset*. (n.d.). https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/pascal.md
31. *Post-training quantization*. (n.d.). TensorFlow. https://www.tensorflow.org/lite/performance/post_training_quantization
32. *Quantization*. (n.d.). Coral. https://coral.ai/docs/edgetpu/models-intro/#quantization
33. *Raspberry Pi 4 model B - Specifications*. (n.d.). Raspberry Pi. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/
34. Raspberry Pi (Trading) Ltd. (2021). *Raspberry Pi 4 Computer Model B*. https://datasheets.raspberrypi.org/rpi4/raspberry-pi-4-product-brief.pdf
35. ruslo. (n.d.). *DeepLab repository*. GitHub. https://github.com/tensorflow/models/tree/master/research/deeplab
36. Saha, S. (2018, Dec 15). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Towards Data Science. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
37. Sarkar, D. (2018, Nov 14). *A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*. Towards Data Science. https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a

38. Saxena, P. (2020, Sep 3). *Increase Frame Per Second (FPS) rate in the Custom Object Detection Step by step*. Towards Data Science. https://towardsdatascience.com/no-gpu-for-your-production-server-a20616bb04bd

39. Shovon, S. (2020, Sep). *Install Ubuntu Desktop 20.04 LTS on Raspberry Pi 4*. linuxhint. https://linuxhint.com/install-ubuntu-desktop-20-04-lts-on-raspberry-pi-4/

40. *TensorFlow*. (n.d.). https://www.tensorflow.org/?hl=es-419

41. *TensorFlow Lite converter*. (n.d.). TensorFlow. https://www.tensorflow.org/lite/convert

42. *TensorFlow Lite Guide*. (n.d.). TensorFlow. https://www.tensorflow.org/lite/guide

43. Torelli, P., & Bangale, M. (n.d.). *Measuring Inference Performance of Machine-Learning Frameworks on Edgeclass Devices with the MLMark™ Benchmark*.

44. *Transfer learning*. (n.d.). Coral. https://coral.ai/docs/edgetpu/models-intro/#transfer-learning

45. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017, May). ImageNet Classification with Deep Convolutional Neural Networks. Communications of the ACM, 60(Issue 6), pp 84–90.

46. *USB Accelerator datasheet*. (n.d.). Coral. https://coral.ai/docs/accelerator/datasheet/

47. *Visual Object Classes Challenge 2012 (VOC2012)*. (n.d.). http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html

48. *What Is Power over Ethernet (PoE)?* (n.d.). CISCO. https://www.cisco.com/c/en/us/solutions/enterprise-networks/what-is-power-over-ethernet.html

49. *What is the Edge TPU?* (n.d.). Coral. https://coral.ai/docs/edgetpu/faq/#what-is-the-edge-tpu

50. YknZhu. (n.d.). *Running DeepLab on PASCAL VOC 2012 Semantic Segmentation Dataset*. GitHub. https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/pascal.md#recommended-directory-structure-for-training-and-evaluation
51. YknZhu. (n.d.). *TensorFlow DeepLab Model Zoo*. GitHub. https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md
52. Yu, F., & Koltun, V. (2016). Multi-Scale Context Aggregation by Dilated Convolutions. *International Conference on Learning Representations (ICLR)*.
53. yukun. (n.d.). *DeepLab Repository*. https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/img/vis3.png
54. Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018). *Shuffle Net: An Extremely Efficient Convolutional Neural Network for Mobile*. Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, pp. 6848-6856.
55. Ultimate Spinach. (1968). *Mind Flowers* (Behold & See)

APÉNDICE A

Configuration Guidelines

This guideline has the purpose to support the users to configure the virtual machine or system intended to execute the semantic segmentation script that has been explained during this work.

The script will run a program in which images captured through a webcam are used to perform semantic segmentation with different models.

Introduction

To begin with, in order to follow these steps it is much easier to work from a Linux environment. For this reason, I will not consider any other Operating Systems. If the user has a Windows computer, it is recommended to install the Oracle VM VirtualBox. I downloaded Ubuntu 20.04 to use it as the Operating System in my virtual machine.

Virtual environment

When Ubuntu is correctly installed, the next step is to create a virtual environment for Python. We install the environment and the Python version needed, in this case python 3.7:

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.7
```

Afterwards, it is necessary to access the directory that the environment is going to be located at, and then create the environment:

```
sudo apt-get install python3-pip
sudo apt install python3-virtualenv python3-venv
virtualenv environment_name -p python3.7.9
```

To activate the environment (from the directory in was created at):

```
source environment_name/bin/activate
```

Tools installation

The next step is to download the corresponding Tensorflow version according to the previously installed Python version. This can be done by using *pip*:

```
pip install tensorflow==1.15
```

In order to use images captured from the webcam as input for the program, OpenCV needs to be installed:

```
sudo apt update
python -m pip install opencv-python
```

To verify that the correct OpenCV version was installed, the following command returns the current version installed in our machine. Output example: *4.2.0*

```
python3 -c "import cv2; print(cv2.__version__)"
```

Webcam configuration

Now that all the necessary tools are installed, it is time to connect the camera to the virtual machine.

When we run our virtual machine, the upper part of the menu shows the window 'Devices' or 'Dispositivos', which should include the option 'Webcams' or 'Cámaras web' once we click on it. At this point, we just need to select our webcam, which should appear listed in that option. If, on the contrary, the option 'Webcams' does not appear in the menu, that means that we need to install the adequate VirtualBox Extension pack corresponding to the version of our VirtualBox.

The final step

Once the webcam is selected, the script should run correctly and obtain the input images through the webcam and semantically segment them.