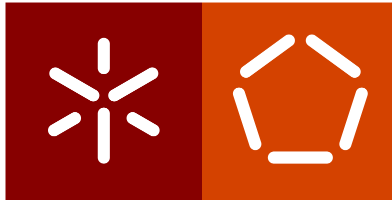
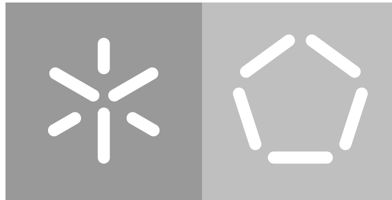


Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Amorim Pereira

Optimization of deep learning algorithms for an autonomous RC vehicle

July 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Amorim Pereira

Optimization of deep learning algorithms for an autonomous RC vehicle

Master Dissertation
Master Degree in Computer Science

Dissertation supervised by
Alberto José Proença
André Martins Pereira
André Leite Ferreira

July 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:

CC BY

https://creativecommons.org/licenses/by/4.0/

ACKNOWLEDGEMENTS

Esta dissertação foi desenvolvida individualmente, no entanto, todo este trabalho foi possível devido a inúmeras pessoas que desempenharam um papel fulcral no desenrolar da minha investigação.

Quero agradecer, em especial, aos meus pais e família, pois sem eles esta dissertação, bem como todo o meu percurso académico envolvente não seria possível. Fizeram sempre de tudo para me fornecer as melhores condições possíveis ao longo de toda esta caminhada, desde o meu primeiro dia académico, até ao último.

Aos meus orientadores, em especial, ao professor Alberto José Proença, que me guiou e auxiliou sempre da melhor forma possível, com excelentes abordagens e com o seu extremo rigor, para que o meu trabalho fosse bem sucedido.

Ao meu supervisor, André Leite Ferreira, que me inseriu na Bosch Car Multimédia, e me auxiliou de muito perto com disponibilidade e prontidão no desenrolar de todo o projeto.

Ao João Fernandes, colega na Bosch Car Multimédia, que me acompanhou e me ajudou em diversas fases da minha dissertação, especialmente em momentos técnicos mais críticos.

Aos meus colegas, Afonso Costa, Joel Morais e Vitor Figueiredo, que também realizaram as suas dissertações muito próximo de mim, na Bosch Car Multimédia, partilhando sempre várias ideias e estratégias.

Aos meus amigos, especialmente aos meus 12 grandes companheiros de Ciências da Computação, que sempre me apoiaram ao longo do meu percuro académico, levando daqui boas memórias e grandes amigos.

Por último, à minha companheira, Ilda Durães, por me ter motivado todos os dias neste projeto, por conseguir ouvir os detalhes, especialmente os obstáculos que foram surgindo nesta dissertação, por acreditar sempre em mim e por me motivar a fazer sempre o melhor.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

This dissertation aims to evaluate and improve the performance of deep learning (DL) algorithms to autonomously drive a vehicle, using a Remo Car (an RC vehicle) as testbed.

The RC vehicle was built with a 1:10 scaled remote controlled car and fitted with an embedded system and a video camera to capture and process real-time image data. Two different embedded systems were comparatively evaluated: an homogeneous system, a Raspberry Pi 4, and an heterogeneous system, a NVidia Jetson Nano. The Raspberry Pi 4 with an advanced 4-core ARM device supports multiprocessing, while the Jetson Nano, also with a 4-core ARM device, has an integrated accelerator, a 128 CUDA-core NVidia GPU.

The captured video is processed with convolutional neural networks (CNNs), which interpret image data of the vehicle's surroundings and predict critical data, such as lane view and steering angle, to provide mechanisms to drive on its own, following a predefined path.

To improve the driving performance of the RC vehicle, this work analysed the programmed DL algorithms, namely different computer vision approaches for object detection and image classification, aiming to explore DL techniques and improve their performance at the inference phase.

The work also analysed the computational efficiency of the control software, while running intense and complex deep learning tasks in the embedded devices, and fully explored the advanced characteristics and instructions provided by the two embedded systems in the vehicle.

Different machine learning (ML) libraries and frameworks were analysed and evaluated: TensorFlow, TensorFlow Lite, Arm NN, PyArmNN and TensorRT. They play a key role to deploy the relevant algorithms and to fully engage the hardware capabilities.

The original algorithm was successfully optimized and both embedded systems could perfectly handle this workload. To understand the computational limits of both devices, an additional and heavy DL algorithm was developed that aimed to detect traffic signs.

The homogeneous system, the Raspberry Pi 4, could not deliver feasible low-latency values, hence the detection of traffic signs was not possible in real-time. However, a great performance improvement was achieved using the heterogeneous system, Jetson Nano, enabling their CUDA-cores to process the additional workload.

KEYWORDS Computer vision, parallel computing, deep learning, inference, homogeneous programming, heterogeneous programming, optimization, autonomous driving.

RESUMO

Esta dissertação tem como objetivo avaliar e melhorar o desempenho de algoritmos de *deep learning* (DL) orientados à condução autónoma de veículos, usando um carro controlado remotamente como ambiente de teste.

O carro foi construído usando um modelo de um veículo de controlo remoto de escala 1:10, onde foi colocado um sistema embebido e uma câmera de vídeo para capturar e processar imagem em tempo real. Dois sistemas embebidos foram comparativamente avaliados: um sistema homogéneo, um *Raspberry Pi 4*, e um sistema heterogéneo, uma *NVidia Jetson Nano*. O *Raspberry Pi 4* possui um processador *ARM* com 4 núcleos, suportando multiprocessamento. A *Jetson Nano*, também com um processador *ARM* de 4 núcleos, possui uma unidade adicional de processamento com 128 núcleos do tipo *CUDA-core*.

O vídeo capturado é processado usando redes neuronais convolucionais (CNN), interpretando o meio envolvente do veículo e prevendo dados cruciais, como a visibilidade da linha da estrada e o ângulo de direção, de forma a que o veículo consiga conduzir de forma autónoma num determinado ambiente.

De forma a melhorar o desempenho da condução autónoma do veículo, diferentes algoritmos de *deep learning* foram analisados, nomeadamente diferentes abordagens de visão por computador para detecção e classificação de imagens, com o objetivo de explorar técnicas de CNN e melhorar o seu desempenho na fase de inferência.

A dissertação também analisou a eficiência computacional do *software* usado para a execução de tarefas de aprendizagem profunda intensas e complexas nos dispositivos embebidos, e explorou completamente as características avançadas e as instruções fornecidas pelos dois sistemas embebidos no veículo.

Diferentes bibliotecas e *frameworks* de *machine learning* foram analisadas e avaliadas: *TensorFlow*, *TensorFlow Lite*, *Arm NN*, *PyArmNN* e *TensorRT*. Estes desempenham um papel fulcral no provisionamento dos algoritmos de *deep learning* para tirar máximo partido das capacidades do *hardware* usado.

O algoritmo original foi otimizado com sucesso e ambos os sistemas embebidos conseguiram executar os algoritmos com pouco esforço. Assim, para entender os limites computacionais de ambos os dispositivos, um algoritmo adicional mais complexo de *deep learning* foi desenvolvido com o objetivo de detectar sinais de trânsito.

O sistema homogéneo, o Raspberry Pi 4, não conseguiu entregar valores viáveis de baixa latência, portanto, a detecção de sinais de trânsito não foi possível em tempo real, usando este sistema. No entanto, foi alcançada uma grande melhoria de desempenho usando o sistema

heterogéneo, Jetson Nano, que usaram os seus núcleos *CUDA* adicionais para processar a carga computacional mais intensa.

PALAVRAS-CHAVE Visão por computador, computação paralela, aprendizagem profunda, inferência, programação homogénea, programação heterogénea, otimização, condução autónoma

CONTENTS

1 INTRODUCTION 1
1.1 Challenges and Goals 2
1.2 Document Outline 3
2 ML INFERENCE FOR AUTONOMOUS DRIVING 4
2.1 Machine Learning Approach to Autonomous Driving 4
2.1.1 Autonomous Driving 4
2.1.2 Computer Vision Techniques 4
2.1.3 Deep Learning 6
2.1.4 Traffic Signs Recognition 9
2.1.5 Frameworks and ML Libraries 13
2.2 Efficient Deep Learning Inference 16
2.2.1 Hardware Support for ML 16
2.2.2 Computational Efficiency 16
2.2.3 Techniques for Efficient Deep Learning Inference 17
2.3 Real-time Evaluation Metrics 19
2.4 The Problem and its Challenges 21
2.4.1 Proposed Approach 23
3 ML INFERENCE ON EMBEDDED DEVICES 26
3.1 The Testbed Environment 26
3.1.1 The RC Vehicle 26
3.1.2 The Arduino Microcontroller 29
3.1.3 The Raspberry Pi and Jetson Nano Embedded Systems 31
3.1.4 The ML Inference to Follow a Path 34
3.2 Setup and Software Installation on the RC Vehicle 36
3.2.1 Setup the Environment 36
3.2.2 The ML Inference to Detect Traffic Signs 38
3.2.3 Performance Evaluation 44
3.3 ML Inference on Raspberry Pi 46
3.3.1 Platforms Decisions 46
3.3.2 System Tuning for ML Inference 46
3.3.3 Challenges 53
3.4 ML Inference on Heterogeneous Jetson Nano 69
3.4.1 Platform Decisions 69

3.4.2 System Tuning for ML Inference 70
3.4.3 Challenges 74
4 EXPERIMENTAL RESULTS 81
4.1 Homogeneous vs. Heterogeneous: a Comparative Evaluation 82
4.1.1 Following a Path 82
4.1.2 Inference Performance of Traffic Signs 90
4.1.3 Traffic Signs: Number of Layers in the Neural Net 93
4.1.4 Traffic Signs: Latency and Throughput 94
4.2 Qualitative Assessment 95
4.2.1 Raspberry Pi vs. Jetson Nano to Follow a Path 96
4.2.2 Detection of Traffic Signs 97
5 CONCLUSION 99
5.1 Homogeneous vs. Heterogeneous: a Comparative Evaluation 99
5.2 Future Work 101

LIST OF FIGURES

Figure 1	Architecture of a artificial neural network	7
Figure 2	Examples of annotations on Cityscapes dataset	12
Figure 3	Part of the Cityscapes code color to represent an object type	12
Figure 4	The traffic sign (left) was detected and segmented (right)	13
Figure 5	Predefined path	21
Figure 6	System architecture	24
Figure 7	Concept Map	25
Figure 8	RemoCar essential components	26
Figure 9	RemoCar chassis parts	27
Figure 10	RemoCar following the track	27
Figure 11	Assembled Remo Car (RC vehicle)	28
Figure 12	Arduino UNO board	30
Figure 13	Wiring connection. Breadboard (center), Arduino (left), board-computer (right)	31
Figure 14	Target embedded systems	32
Figure 15	CNN architecture	35
Figure 16	Input image example	35
Figure 17	Segmentation of traffic signs. Attribute (left), target (right)	39
Figure 18	Isolation of traffic signs segmented areas	40
Figure 19	Parallel architecture to run both DL workloads	40
Figure 20	Output of traffic signs inference: masks of predicted traffic signs.	41
Figure 21	ROIs of the predicted masks from Figure 20	42
Figure 22	Dimension of the ROIs computed in Figure 21	42
Figure 23	ROIs of predicted images with poor results	43
Figure 24	Autonomous benchmark script architecture	45
Figure 25	The three phases of the inference algorithm	47
Figure 26	Inference phase: steering angle prediction	48
Figure 27	Post-training quantization	48
Figure 28	Communication process between Python and C++	51
Figure 29	TensorFlow and TensorFlow Lite inference times with different file weights	54
Figure 30	Arm NN inference times with different file weights	55
Figure 31	Arm NN inference times with different file weights	56

Figure 32 Arm NN inference times with different file weights 57
Figure 33 Arm NN inference times with different file weights 58
Figure 34 Data pipeline of Arm NN engine 58
Figure 35 Inference times of multiple libraries including Arm NN 64 bits versions 59
Figure 36 Modified neural net (*1-oconv*) 60
Figure 37 Modified neural net (*2-odrop*) 60
Figure 38 Modified neural net (*3-drop_conv1*) 61
Figure 39 Modified neural net (*4-drop_conv3*) 61
Figure 40 Inference times of tuned neural networks 62
Figure 41 Profile of *2-odrop* neural network 62
Figure 42 Profile of *3-drop_conv1* neural network 63
Figure 43 Performance of the developed implementations 64
Figure 44 Samples of 1 second of system memory consumption. Purple box: the moment when the system rans out of memory 66
Figure 45 Bug on output of traffic signs inference 66
Figure 46 Inference on FgSegNet with *instance_norm* 67
Figure 47 Inference on FgSegNet with Batch Normalization 68
Figure 48 Inference on the original FgSegNet (with Instance Normalization) 68
Figure 49 A convolutional neural network without any optimizations 72
Figure 50 The convolutional neural network after several optimizations applied by TensorRT engine 72
Figure 51 Discrete GPU architecture 73
Figure 52 Integrated GPU architecture 73
Figure 53 Performance of Jetson Nano TensorRT against Raspberry Pi implementations 74
Figure 54 Correct output values predicted by TensorFlow, TensorFlow Lite and Arm NN 75
Figure 55 TensorRT wrong predicted values 75
Figure 56 TensorRT conversion pipeline 76
Figure 57 Single output neural net w/ a convolut layer 78
Figure 58 Basic neural net with two output neurons 78
Figure 59 Single convolutional layer followed by two output neurons 78
Figure 60 CNN without the convolutional and the max pooling layers 79
Figure 61 CNN without convolutional layers 79
Figure 62 TensorRT error while parsing the FgSegNet model 79
Figure 63 FgSegNet prediction using TensorRT 80
Figure 64 Autonomous driving of the RC vehicle 81

Figure 65 Inference times on Raspberry Pi 4 82
Figure 66 RAM Memory consumption on Raspberry Pi 4 84
Figure 67 Inference times on TensorRT (half-precision vs. single-precision) 85
Figure 68 Inference times on TensorRT (half-precision vs. single-precision), using 256x192 image resolution 86
Figure 69 RAM Memory consumption on Jetson Nano. 87
Figure 70 TensorRT inference executions varying the upper-limit of memory usage. 88
Figure 71 Overall comparison of inference times among both embedded systems 88
Figure 72 Overall comparison of execution times among both embedded systems 90
Figure 73 Overall evaluation of traffic signs inference on multiple libraries 91
Figure 74 Traffic signs inference time on TensorRT using half and single-precision 92
Figure 75 Memory consumption of traffic signs detection algorithm on multiple libraries 93
Figure 76 Inference time of generated submodels 94
Figure 77 Prediction of an uncontrolled environment images using FgSegNet 98
Figure 78 Environment of predefined track of RC vehicle 98
Figure 79 Example of a frame recorded by the RC vehicle 98

LIST OF TABLES

Table 1 Recognizable objects by Cityscapes dataset 11
Table 2 Reaction time of driver Li et al. (2019) 20
Table 3 Raspberry Pi 4 hardware specifications 32
Table 4 Jetson Nano hardware specifications 33
Table 5 Jetson Nano Graphics Processing Unit (GPU) specifications 34
Table 6 Supported power modes of Jetson Nano 34
Table 7 TFLite post-training quantization options 49
Table 8 Operating systems and profiling tools supported by Flame Graph visualizer tool 56

ACRONYMS

ACC Adaptive Cruise Control. 20
AI Artificial Intelligence. 1, 4, 6, 13, 22, 37
ANN Artificial Neural Network. 1, 7, 8
API Application Programming Interface. 13
ARM Advanced RISC Machine. 2, 15, 17, 22, 23, 32, 33

BLAS Basic Linear Algebra Subprograms. 9

CNN Convolutional Neural Network. 8, 9, 17, 18, 22, 25, 34–36, 40, 41, 47, 57, 59, 61, 64, 69, 76–78, 86, 96
CPU Central Processing Unit. 15–18, 21–24, 32, 33, 57, 71, 72, 78, 79, 88, 89
CUDA Compute Unified Device Architecture. 13, 14
CV Computer Vision. 5

DL Deep Learning. 1, 2, 11, 12, 14–17, 21–23, 25, 33, 37–39, 43–46, 64, 65, 69, 74, 78, 88–91, 93, 94, 99–101
DNN Deep Neural Network. 1, 18, 23
DRAM Dynamic Random Access Memory. 18

ESC Electronic Speed Control. 26, 27, 29, 31

FLOPS Floating-point Operations Per Second. 9, 89
FMA Fused Multiply–Add. 17
FP16 Half-precision Floating-point. 13, 15, 33, 49, 70, 71, 85, 86
FP32 Single-precision Floating-point. 13, 71, 85, 86, 92, 100
FPS Frames Per Second. 82, 83, 89, 91, 95, 100
FPU Floating-point Unit. 32

GFLOPS Giga Floating-point Operations Per Second. 33, 34
GPU Graphics Processing Unit. vi, 2, 9, 13–17, 22–24, 31–34, 70–72, 79, 85, 86, 88, 89, 91

HW Hardware. 17, 23, 24

IC Integrated Circuit. 31
IDE Integrated Development Environment. 29, 30
INT8 8-Bit Integers Precision. 14

ML Machine Learning. 1, 2, 12–14, 22, 31, 39, 55, 57–60, 64, 69, 71, 94, 101

NN Neural Network. 15, 23

NPU Neural Processing Unit. 15, 16

OS Operating System. 44, 65, 87, 92

RAM Random Access Memory. 32, 33, 65, 66, 68, 84, 87, 89, 92, 93, 100

RC vehicle Remo Car. iii–v, 2, 12, 16, 21, 22, 26–29, 31, 37, 40, 42, 43, 46, 47, 49, 53, 55, 59, 70, 81, 83, 89, 91, 95–101

ROI Region of interest. 42, 43

SIMD Single Instruction, Multiple Data. 17, 19, 32, 50, 88

1

INTRODUCTION

The automotive industry is changing with the arrival of an emerging mobility value chain that is being actively studied and developed by companies nowadays - the autonomous driving. This topic is becoming more present and with many interests, which led many companies to develop hardware and software technologies towards to a fully autonomous driving capability.

Artificial Intelligence (AI) is the key to self-driving cars and, therefore, Deep Learning (DL), a sub-domain of Machine Learning (ML) algorithms based on Artificial Neural Networks (ANNs), has been proving to be quite efficient on problem solving of tasks such as object detection, speech recognition, language translation and others. This computing field is being increasingly explored to provide an autonomous and less complex way to solve these challenges.

Multiple types of ANNs, such as Deep Neural Networks (DNNs), have been successfully applied to this subject, since vision-based navigation of a vehicle requires a high-quality and precise level object detection, which certain types of neural networks can deliver.

This scientific field — computer vision — plays a key role in autonomous driving, since it seeks to automate tasks that humans can do by giving the capability to the machine to take decisions by analysing, processing and extracting high-dimensional data from digital images that represent the real world. Relevant environment data can be captured through embedded sensors and systems attached to the car, so this information can be processed in a way to derive intelligent vehicle control output, such as the steering angle and the acceleration to safely drive the car.

To successfully deploy this know-how in the car, embedded systems are required. All manufactured cars, nowadays, already contains hundreds of embedded systems that most people are not aware of. Climate control, infotainment panel or built-in safety systems are all controlled by these systems. And in the future self-driving cars will be equipped with a vast and numerous of embedded systems needed for additional computation and tasks distribution, absolutely crucial to build an autonomously driven car.

Processing these AI workloads on embedded computing platforms adds up new challenges since these platforms are not built to process big and heavy workloads. However, these intelligent driving vehicles carry core algorithms that are computationally demanding and there

is the necessity to have a good real-time processing flow. Therefore, computing power has to satisfy the needs of producing intelligent output data from the car components/embedded systems so it can derive an autonomous control. Hence, choosing the right hardware is not a logical and linear option since it must also fit additional constraints like cost, size and power consumption.

1.1 CHALLENGES AND GOALS

This dissertation is aligned with the autonomous driving work being developed at Bosch Car Multimedia, S.A. And the introduction of this concept into the company's development pipeline is an asset, which requires a careful selection of optimized embedded systems and the right ML tools to efficiently deploy the deep learning algorithms. The challenges for this work also includes the optimization of DL algorithms running on different fine-tuned frameworks, which will be tested in a 1:10 scaled remote controlled car.

DL algorithms are computational intensive, not only during the training of the neural networks but also in the inference phase, due to the number of parameters and the required number of layers being processed. In monitor/control systems with real-time requirements, such as object detection in autonomous vehicles, an embedded computing platform with GPU cores as accelerators may help to provide the required performance.

Bosch supplied the RC vehicle to test, validate and improve the performance of existing DL implementations. The RC vehicle is remotely configured and uses an embedded hybrid system with Advanced RISC Machine (ARM) cores to autonomously monitor and control the RC vehicle, with some predefined constraints. The evaluation of the inference performance of several DL algorithms running in this RC vehicle considered two types of embedded systems: homogeneous or heterogeneous with on-chip computing accelerators (GPU cores).

The main goals defined for this dissertation were:

1. To deploy and evaluate DL algorithms on two different Linux embedded systems:
   - the Raspberry Pi 4 (with a quad-core Cortex-A72);
   - the NVidia Jetson Nano (with a quad-core Cortex-A57 and a 128 CUDA-core NVidia GPU Maxwell).
2. To improve the inference performance of the algorithms by enabling hardware and software optimizations, exploring the following ML libraries and frameworks:
   - TensorFlow;
   - TensorFlow Lite;
   - Arm NN;
   - PyArmNN;

- TensorRT.

1.2 DOCUMENT OUTLINE

This document is structured in five chapters. The first one, this **Introduction**, presented the work context, the challenges and the work goals. Chapter 2, **ML inference for autonomous driving**, describes the state of the art of the relevant information to understand the problem and to allow the search for an innovative and efficient solution. Chapter 3, **ML inference on embedded devices**, describes the testbed environment with full details of the embedded systems in the RC vehicle, with a focus on system tuning for ML inference and on the developed implementations and its challenges. Chapter 4, **Experimental results and discussion** details and discusses the evaluations and tests performed on the developed implementations, presenting a qualitative and a quantitative analysis of the delivered outcomes. Chapter 5, **Conclusion**, gives an overview of the developed work, critically summarising the core work of this dissertation and giving suggestions for additional features that may enrich the work of this project.

2

ML INFERENCE FOR AUTONOMOUS DRIVING

2.1 MACHINE LEARNING APPROACH TO AUTONOMOUS DRIVING

2.1.1 *Autonomous Driving*

Developing a full self-driving car means to create a complex and intelligent system capable to handle complex traffic scenarios in urban and highway environments at any time. This complexity calls for the artificial intelligence field since its goal is to provide cognitive abilities to machines, and on this use-case can processes an high amount of data efficiently, train it and validate the autonomous driving systems.[1]

Nowadays, one of the biggest challenges for the experts is to improve the efficiency of the systems intelligence, which must understand all the surrounding environment and all moving and static objects, as well as predict the next move of the vehicle in order to adapt itself to a variety of circumstances that may occur ahead on its route.

Autonomous driving requires a wide range of information and knowledge and needs to understand the environment, determine the exact position and decide how it should behave towards a given condition on the road. Thus requiring AI and computing power to provide an advanced system to safely handle complex traffic situations and rely upon several conditions.

2.1.2 *Computer Vision Techniques*

This scientific field deals with the high-level understating of digital images or videos, concerning with automatic extraction, analysis and understanding of useful information from images. In such way, this technology seeks to automate tasks performed by the human visual system Szeliski (2011).

Object detection and classification are two important tasks that play a major role in the autonomous driving, in a such way that the car needs to record and process the image of the vehicle route to track the environment and recognize the path that it must follow and the

1 Interview with Moritz Dechant in 2018 on "Self-driving car technology — Between man and machine", in https://www.bosch.com/stories/autonomous-driving-interview-with-moritz-dechant/.

obstacles that must avoid. Because computer vision can extract an high-level of understanding from digital images or videos, it delivers great techniques and algorithms to accomplish these desired tasks, since it deals with autonomous planning for navigation systems.

This technology has been seen playing a big role in the automotive field, since the 90s, with the automatic car plate recognition, among other areas. But for this scenario, an efficient and fine-tuned collection of technologies must be used in the future to fully promote an advanced intelligent system capable to self-drive vehicles.

Traditional Approaches

Computer vision algorithms work, in general, by extracting features from images and using them to classify and find patterns on them. Despite the recent use and investigation of these techniques, computer vision began in the late 1960s, pioneering artificial intelligence.

Multiple vision related tasks were implemented, mostly recurring to the extraction of multiple features from images, such as edges, corners, colors, letters, numbers, deemed to be extremely relevant in such tasks. Lee (2016).

Traditional techniques for image matching and feature detection were implemented with different feature-based approaches such as, SIFT (Scale-Invariant Feature Transform) Karami et al. (2017), SURF (Speeded-Up Robust Features) Bay et al. (2006), BRIEF (Binary Robust Independent Elementary Features) Michaeland et al. (2010), Features from Accelerated Segment Test (FAST) Rosten and Drummond (2006), Hough transforms Goldenshluger and Zeevi (2004), geometric hashing Tsai (1994), among other wide-range of conventional Computer Vision (CV) algorithms.

The difficulty of these approaches on feature extraction or image classification is that features must be explicitly chosen, in a such way that the algorithm may trace them on given input images. The problem, is that becomes hard to cope with CV algorithms when the number of classification classes or features to extract from an extremely high number of images increases. So when comparing to modern algorithms, these old methods can provide a better performance, however the accuracy and the precision becomes degraded of the predicted the data.

So this traditional approach is not the most suitable one to support the process of feature detection of real-time data that requires an high value of accuracy, so the vehicle can safely drive through the path.

Artificial Intelligence Approach

Until recently, computer vision only worked with limited capacity due to the limitations of the required computational power for these workloads. Over the last few years, major efforts and advances in both machine learning algorithms and computer hardware had been pushed towards. This led to the application of computer vision techniques that already existed years

ago, but it was not possible to deploy them on the old machines because it was lacking the resources to handle these heavy AI workloads.

These advances in artificial intelligence and innovations in deep learning and neural networks has been able to surpass humans in some tasks related to detecting and labeling objects.

One of the driving factors behind the growth of computer vision is the increasing amount of accessible data, since this technology needs big data. The increasing amount of information is leading to the extraction of even more knowledge, which leads to better predictions since the training datasets are becoming richer in information.

This approach, using modern AI techniques may achieve substantially better accuracy, due to the presence of an higher amount of information that can be processed on the latest devices that can handle this amount of mathematical operations existing on these algorithms O'Mahony et al. (2019).

2.1.3 *Deep Learning*

Deep Learning is a particular type of machine learning, which uses artificial neural networks to make intelligent systems. This is, trains computers to perform tasks as human behaviours do, which includes, speech recognition, image recognition and prediction of data. Instead of organize the data to be executed through predefined methods and algorithms, deep learning configures basic parameters upon the data and trains the computer to learn by itself through patterns recognition on multiple processing layers. This type of AI is widely used in computer vision, speech recognition or natural language processing tasks because it can classify, cluster and predict data with a good accuracy. Goodfellow et al. (2016)

Deep learning architecture is characterized by making use of artificial neural networks, which are trained by analysing an amount of data. This learning process can be supervised, semi-supervised or unsupervised. The algorithms are often trained with a large amount of data Géron (2017), hence this intense process must be executed, in most cases, by powerful server machines/workstations.

This computing area holds on a powerful set of technologies and techniques that achieves great power and flexibility by learning to represent the world, driving to the biggest innovations in the most diverse areas such as computer vision, natural language processing, healthcare and genomics Zhang et al. (2020).

The usage of deep learning algorithms is usually comprehended into two stages, training and inference. Training is the process where the neural network processes the input to map the original input into a such classification or into a quantification of some variable or data analysis.

Artificial Neural Networks

ANNs design were inspired from biology, group theory and a lot from research and constitutes sets of algorithms intended to recognize patterns.

Artificial neural networks are composed by neurons, synapses, weights, biases and functions. A neuron is the basic unit of the neural network that receives data, processes it and transmits it.

These networks can have millions of neurons connected, which makes this system capable to analyse and find deep patterns on different kind of information, extracting knowledge from them.

As presented in Figure 1, the neurons are divided into three groups: (i) input neurons that are responsible to receive the input data of the algorithm; (ii) hidden neurons that processes information transmitted from previous layers; (iii) and output neurons that produces the output data.

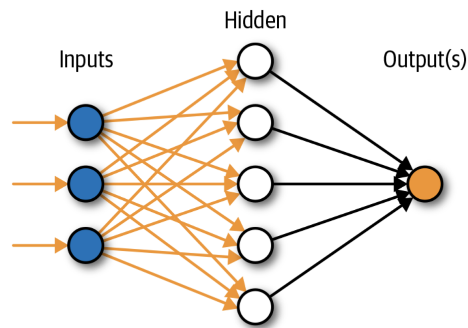


Figure 1: Architecture of a artificial neural network

Neurons can only operate on numbers between 0 and 1 or between -1 and 1, so to translate the input data into numbers readable by the neurons, a normalization process must be executed. This process is responsible for analysing the input data, map it to numbers and then convert them to a range that is understandable by the neurons.

Synapses are the connections between the neurons, and the role of them is to learn and to memorize patterns of the input data. Each synapse will hold a weight, which adds up changes to the input information. Weights plays an important role to predict data, since they use numerical parameters to determine the effectiveness of neurons by using them as input and mapping them to different values to be used by other neurons on next layers.

The networks can be trained recurring to two learning techniques: (i) supervised learning; and (ii) unsupervised learning.

In supervised learning, a training dataset is given to the model. This training data is represented by multiple input examples and the corresponding target outputs. When the network is learning from the input data, the weights are constantly adjusted for error reduction in order to map input data into different values that can be used by another neurons, passing

by more weight synapses in order to replicate the desired output. The network iterations above this process run multiple times until the weights are correctly adjusted to compute the desired result. In supervised learning, the training process must be executed to generate a network able to predict data.

Unsupervised learning does not have a training dataset that contains input data that must be mapped into target outputs. During this process, the ANN network adjusts its weights by processing input data, producing an output similar to the input. The model is able to recognise the patterns and the differences of the inputs, without any external data examples. In this process, clusters, which are a group of multiple weights, are formed, on which each cluster is able to find a certain pattern on the input data.

Convolutional Neural Networks

Most modern deep learning models are based on artificial neural networks, specifically, Convolutional Neural Networks (CNNs). And are widely used for image classification, face recognition, scene labeling and more.

CNN is based on the animal's visual cortex and is a specialized kind of neural network for processing data that assumes a special spatial structure on its input, designed to map image data to an output variable. This convolutional networks are designed to work with two-dimensional image data.

Traditional ANNs were not built for image processing because its neurons are not placed like the brain area structure responsible for processing visual stimuli in humans or other animals. Thus CNNs have proving to be highly efficient on image data related prediction problems.

The neurons in the CNNs present in the hidden layers are only linked with a subset of neurons of the preceding layer. This connectivity architecture permits the CNN models to learn discreet features from input data.

The convolutional name comes from the convolutional layer, which performs an operation called ”convolution”, that is always present in these types of networks.

A convolution is a linear mathematical operation defined by the multiplication of a set of weights with the input. Because this technique was designed for two-dimensional input data, the multiplication is processed using an array of input data and a two-dimensional array of weights.

A particularity of the CNNs, that are an extension of artificial neural networks, is the presence of the convolution operation, whose purpose is to extract useful features from the input. In image processing, a wide range of different filters is available to use for convolutions. Each one helps to extract different data and features from the input image data Dumoulin and Visin (2016).

Convolutions are applied to extract useful features from the input. In image processing, there is a wide range of different filters one could choose for convolution. Each type of filters helps to extract different aspects or features from the input data such as an image.

However, most convolutional networks are computationally intensive due to the high amount of data present in the input which will be mapped to multiple layers, where each one contains neurons. This high number of neurons will be accessed multiple times across the training stage, and multiple mathematical operations will be processed targeting to compute the final weights of the network, which means the network was trained.

Predictive performance comes when exploring and optimizing these nets as they tend to require fewer parameters than dense architectures, so they carry a smaller density. Also the mathematical convolutions operations applied provides an easy approach to parallelize its work across GPU-cores Szegedy et al. (2015).

Computation on CNNs

Basic Linear Algebra Subprograms (BLAS)-3 operations performed on CNNs can be both math-limited or memory-limited, depending on multiple factors such as: matrices sizes, batch size, CNN type-model, etc. For example, a fully-connected layer applied to a single vector (with a tensor of mini-batch size 1) is memory limited.

If there is a bigger batch size, the operations number pushed rises up, but also the chances to parallelize the problem increases. So when exploring both hardware and software parallel techniques, eventually it will be possible to get more Floating-point Operations Per Second (FLOPS) during (training and) inference phase.

So the arithmetic processing intensity, more exactly the floating point arithmetic, will be delimited by multiple factors, being in most cases limited (FLOPS limitation) by two main bottlenecks:

- **CPU-bound:** element-wise operations (activation function: relu, sigmoid, Tanh, etc.), reduction operations (pooling and normalization layers, SoftMax).

- **Memory-bound:** most operations are memory-limited, since nowadays datasets are not small, performing little operations per byte accessed.

2.1.4 *Traffic Signs Recognition*

Automatic traffic sign detection and recognition is becoming more and more present in our lives, playing an important role in advanced driver assistance systems and automated driving, which is already a reality we are living it.

Recognizing road signs automatically is essential for the automotive industry's efforts since this advanced process may help the drivers by supporting them to easily and safely drive

and guide a vehicle Ciregan et al. (2012). Asides it can also warn the driver of their actions on crowded streets and even on zones that suffers multiple speed limit variations.

Multiple studies have already been carried out running on a machine learning approach to solve this problem: several types of neural networks were already developed and tested to solve this use-case Maldonado-Bascón et al. (2007), including efforts to efficiently recognize, in real-time, text of traffic signs Greenhalgh and Mirmehdi (2015).

However, this automated task is not computationally easy because of the wide variations of traffic signs appearance due to partial occlusion, different viewpoints or weather conditions, among others Luo et al. (2018). This carries a high volume of data that must be given to algorithms, most of them relying on neural networks. The system of neural networks can easily achieve millions of neurons on its composition, adding up even more complexity when combining the data with these vast mathematical operations.

Recently, large amounts of datasets containing huge amounts of real images of traffic signs on different environments have been publicly released. The presence of this large information enables the possibility to train multiple traffic signs algorithms and efficiently recognize them.

These large public datasets contains substantial amounts of the different types of traffic signs recorded in several cities and environments. This grow of the release of public traffic signs data and the amplification of the signs variety on the datasets are crucial for the learning process of machine learning models.

On this project, it was used a large-scale dataset containing information and tracking of multiple road journeys on a vehicle, which contains useful information to proceed to the segmentation and classification of objects that are present when driving a vehicle. On this project it urged the need to detect European traffic signs in real-time, hence the usage of *Cityscapes* dataset to process and train a developed neural network capable to automate this process: detect traffic signs.

Cityscapes Dataset

Traffic signs recognition requires a large-scale dataset containing a vast amount of information regarding to multiple environments and weather conditions containing traffic signs and the segmentation of them. It is important the dataset yields a magnitude of decisions on all recorded and postprocessed data, and that offers an annotation protocol capable to identify segmented objects on each image.

Cityscapes can provide these prerequisites, offering a semantic understanding of complex urban street scenes. This dataset is composed by a large number of high-quality images, recorded in streets from 50 different cities. On total, this dataset aggregates about 25,000 annotated images with dense semantic segmentation and instance segmentation of vehicles, people and some objects or obstacles, such as buildings or trees.

All objects were captured on multiple seasons and daytime, capturing the same object type on different environment circumstances, such as different illumination and weather conditions. An extensive metadata is also provided with the corresponding right stereo views, GPS coordinates, ego-motion data from vehicle odometry and even outside temperature from vehicle sensor Cordts et al. (2016).

The dataset offers an extensive complexity of groups that can be detected, such as vehicles, constructions, multiple objects and even flat types. Inside these groups, each segmentation is also categorized into a classe definition, therefore it can distinguish multiple object types that belongs to a certain group. Table 1 presents all types of objects the Cityscapes can recognize, categorized into the Class notation. Each instance of a labeled Class may also belong to a Group that aggregates multiple instances of the same type.

Group	Classes
flat	road, sidewalk, parking, rail track
human	person, rider
vehicle	car, truck, bus, on rails, motorcycle, bicycle, caravan, trailer
construction	building, wall, fence, guard rail, bridge, tunnel
object	pole, pole group, traffic sign, traffic light
nature	vegetation, terrain
sky	sky
void	ground, dynamic, static

Table 1: Recognizable objects by Cityscapes dataset

Each recorded image may have attached different types of annotations, segmentation or even have distinct metadata categories Savkin et al. (2020). For this use-case, only object detection is required, therefore only segmented images are going to be used. These images will serve as training data so the DL models can be trained, providing them the necessary input data to train the networks to be able to detect traffic signs.

Cityscapes dataset visually distinguishes multiple object types by annotating them with a defined color scheme. Figure 2 displays an example where two different images of Cityscapes were processed to generate the right annotations of the instances present on the road Imam et al. (2019). The left images are the original ones recorded, the right ones are the equivalent from the left of each other, where the main objects and instances were segmented, filling them with different colors.

Each supported Class is painted on a code color following a color protocol defined by Cityscapes, representing an object type. Each object is segmented on a specific color and attributed to a certain Class. For example, all segmented traffic signs are painted on yellow, all detected persons are painted on red, and so on. Figure 3 presents part of the code scheme defined by Cityscapes to represent different object types.

Figure 2: Examples of annotations on Cityscapes dataset

Road	Sidewalk	Car	Pole	Building	Sign	Fence
Tram	Vegetation	Static	Sky	Wall	Dynamic	Person

Figure 3: Part of the Cityscapes code color to represent an object type

Although the chosen dataset has the possibility to recognize multiple objects, the RC vehicle needs only to be aware of traffic signs. Therefore only the yellow color — that represents traffic signs — will be analysed by the algorithms. The major disadvantage on this color scheme strategy to define traffic signs is the impossibility to categorize each traffic sign. The algorithm should easily detect a traffic sign but will not have the ability to distinguish a stop sign from a left sign.

The algorithm that is used to run the traffic signs detection is also based on neural networks, more precisely a network called *FegSegNet* that will be described next.

FgSegNet

While the RC vehicle is driving through the predefined path, it was decided that it must process each frame in order to also predict the existence of traffic signs along the predefined path the vehicle is running.

To accomplish this desired behaviour, a robust ML model of foreground segmentation is required. As Bosch Car Multimédia, S.A. targets its efforts to automotive industry, an additional work was accomplished that aimed to develop a DL model that segments moving objects from the background under different challenging scenarios.

The provided developed AI algorithm follows the previous work of Lim and Keles (2019) on Foreground Segmentation Network (aka FgSegNet), which uses a triple convolutional neural network for multi-scale feature encoding the foreground segmentation. It extracts foreground objects from video sequences or real-time images.

FgSegNet is the ideal neural network to successfully find objects of given input images. The model processes its input and extract objects of interest, generating an output image that only contains a segmentation of the found object. An example of this process is displayed in Figure 4.

Figure 4: The traffic sign (left) was detected and segmented (right)

2.1.5 *Frameworks and ML Libraries*

To deploy the deep learning algorithms, four different machine learning frameworks or libraries were used: TensorFlow, TensorFlow Lite, TensorRT, Arm NN and PyArmNN.

These tools can enable specific hardware optimizations, like enabling ARM-based multi-threading and multi-processing abilities, engage GPU-cores or even compute inference on Half-precision Floating-point (FP16) or Single-precision Floating-point (FP32).

TensorFlow

TensorFlow is an open source library written in Python and C++, for numerical computation, created by Google. Machine learning algorithms can be easily created and deployed, facilitating the beginners to start with, as it is flexible and reliable, and provides a full documentation stack. This library can smooth complex ML tasks, such as the data acquisition process, the model training, inference, and even on fine-tuning the model.

It uses Python to provide a user-friendly Application Programming Interface (API) to build machine learning applications, but its real engine is executed in high-performance C++. It also supports NVidia GPU processing by providing support to Compute Unified Device

Architecture (CUDA) extension. The drawback is the limited number of supported GPU devices, it mostly support only NVidia platforms.

Numerical computations are expressed as stateful dataflow graphs, whose vertices are represented by operations and edges by tensors, which can represent input images. These graphs can be analysed and optimized before execution, offering multiple possibilities to improve its performance. It also has been proving to be energy efficient on multiple research studies.

One of the major current weaknesses of the framework is the amount of time that needs to construct a new DL architecture. As a result, constructing advanced and complex deep architectures is not very convenient since it will generate a dynamic structure as all produced models Zadeh (2018).

TensorFlow Lite

TensorFlow Lite (TF Lite) is a Google open source deep learning framework, only for on-device inference.

This framework is highly optimized to deploy DL models on mobile and embedded devices, providing tools to optimize the size and performance of models, such as data quantization and compression techniques, often with minimal impact on accuracy.

It gives the ability to run inference on limited-resources embedded devices, by converting TensorFlow pre-trained loaded models into TensorFlow Lite format. TF Lite models cannot be trained again but it provides model optimization techniques like pruning. Therefore, TensorFlow Lite it is not an extension of TensorFlow, not supporting all its full operations and capabilities, since TF Lite is inference oriented only, so models must be externally created and trained using another library or framework.

Briefly, its architecture is defined by four sequentially steps: model picking, conversion, deployment and optimization. A pre-trained model must be first uploaded to this inference framework, to be later compressed to the TF Lite format. Then this model is deployed on the target embedded device, applying some optimizations, such as quantization, by converting 32-bit floats to 16-bit Bfloats or 8-bit integers, or even run on a separate accelerator device (e.g., a GPU).

TensorRT

TensorRT is a NVidia framework for high-performance deep learning inference, that enables CUDA cores of NVidia GPUs.

It includes a DL inference optimizer and runtime, delivering low latency and high-throughput for deep learning inference applications (NVidia, 2019).

TensorRT is built on CUDA, NVidia's parallel programming model, and enables inference optimizations for loaded models trained in a wide-range of ML frameworks. 8-Bit Integers

Precision (INT8) and FP16 optimizations are available for production deployments of DL applications such object detection on automotive industry.

After applying optimizations, TensorRT can fine-tune its model by selecting platform-specific optimized kernel for specific NVidia systems, such as Jetson embedded platforms.

With support for every major framework, TensorRT helps process large amounts of data with low latency through powerful optimizations, use of reduced precision, and efficient memory use.

Arm NN

Arm NN is an inference engine for Central Processing Units (CPUs), GPUs and Neural Processing Units (NPUs). It fills the gap between the existing Neural Network (NN) frameworks and the embedded devices architecture, yet bounded to ARM-series products only.

It offers an efficient translation of existing NN frameworks, such as TensorFlow and Keras. And allows inference to run efficiently across Arm Cortex-A CPUs, Arm Mali GPUs and Ethos NPUs.

This open-source Linux software tool, written using portable C++14, allows machine learning workloads to be deployed on power-efficient devices, supporting all models trained on a vast number of commonly used frameworks.

Arm NN library is built on top of Arm Compute Library that leverages NEON acceleration instructions on ARM CPUs and OpenCL acceleration on Arm Mali GPUs.

This NN inference-only tool is the newest ARM technology release, accelerating its produced devices while running deep learning inference workloads.

PyArmNN

PyArmNN is a Python extension for the earlier described inference engine, Arm NN. It provides a similar interface to Arm NN C++ Api and it is built around the earlier library.

PyArmNN does not implements any computation kernels, all processing is instructed by Arm NN. So this Python extension it is simply an interface to access the specified inference engine but using another programming language. It also manages and manipulates the memory being used differently.

2.2 EFFICIENT DEEP LEARNING INFERENCE

2.2.1 *Hardware Support for ML*

Target systems

This thesis work aims to improve the performance of a computing system while running deep learning algorithms to autonomously drive a RC vehicle. When building this system, after the very first prerequisite — system performance — the system may also fit in a limited space as it will be placed on a small remote controlled car.

Since the RC vehicle car is a simulation of an actual vehicle of the real world, price also enters into the full-stack pre-requisites that the system must satisfy in order to be built. Like so, two systems were proposed by Bosch (the company funding this dissertation work), which is betting on machine learning techniques, delivering this technology to real world market as automotive industry.

Raspberry Pi 4 is the cheapest proposed embedded system, while Jetson Nano, with an integrated GPU accelerator, is the one capable to deliver more computational power. Subsection 3.1.3 presents these systems in more detail.

2.2.2 *Computational Efficiency*

Embedded devices are getting more suitable to be deployed with complex workloads whose hardware could eventually handle but there is the need to fully activate its limited resources and manage memory consumption. Also the huge diversity of embedded systems does not aid on the process of deploy a single deep learning algorithm efficiently across the wide range of existing CPU architectures. Some platforms also provide additional accelerator units, such as GPUs or NPUs, which could be enabled. Optimizations on hardware-level are described, which can be managed by inference engines or frameworks.

Efficient Deep Learning

Jiang et al. (2018) present in their paper two-staged optimizations to tune DL models on a wide range of CPU architectures since frameworks cannot always tune it to a specific CPU or GPU model.

The first stage is concerned to optimizations on computation graph, which is generated to process the workload. The goal is to transform a computation graph into a minimized one to reduce execution time and memory consumption. The techniques applied has major similarities to compiler optimizations: **constant folding**, **graph simplification**, **kernel fusion**, **pre-computing layout transformation** and **quantization**.

The second stage explores specific algorithms to achieve an output such as convolutional implementations, selecting the ones that take advantages of hardware capabilities, namely **tiling**, **reordering**, **loop unrolling**, **vectorization**, **parallelization**. The best combinations depends on the hardware specifications.

TVM (2016) was used to develop this work and a compiler stack for DL to achieve the best performance among frameworks and Hardware (HW) backends.

Parallelizing Multiple CPUs/GPUs to Speedup DL Inference

Wang (2019) introduce in his paper several ways to parallelize inference, focusing on data parallelism by equally splitting the input data among the available CPU/GPU-cores.

Among the different techniques presented, the most relevant one described on his work was recurring Python multiprocessing. This technique uses multiple threads to take advantage of the multiple on-chip cores, achieving parallelism by concurrently running DL inference on multiple cores.

When applied to this dissertation, each captured frame is sent to a different core, so multiple frames can be simultaneously processed. The only drawback is that we must be careful with out-of-order executions, since when capturing video the frames cannot be mounted again unsorted.

Optimizing CNN Model Inference on CPUs

Liu et al. (2019) proposed on their work a solution proving an efficient inference while compiling and optimizing CNNs on several CPUs, one of them the ARM Cortex-A72.

The experimental results provided a speedup up to 3.45x on the multiple CNNs models used, compared to the performance of the state of the art solutions.

Several CPU features are explored such as Single Instruction, Multiple Data (SIMD), Fused Multiply–Add (FMA) and parallelization to optimize the nets, by managing its implementation in a high-level.

The data layout is first adequately structured to reduce memory access overheads, leveraging the FMA operation utilization. Then, there is dimension reordering to use the 128-bit vectorization extension instructions in an ARM device (NEON).

2.2.3 *Techniques for Efficient Deep Learning Inference*

Deep Learning is becoming more and more popular nowadays and with the increased amount of data, neural networks are getting larger, which increases the workload intensity overkilling the processing units, specially limited ones, such as in embedded systems.

As the model size becomes larger, it becomes more difficult to be deployed on embedded devices, which may lack memory capacity.

Running neural networks on resource-limited platforms requires efficient solutions with ”algorithms and hardware co-design”. Next are presented several solutions from data engineering and data science that optimizes neural networks architecture, improving its performance.

Deep Reuse

Deep Reuse is a technique presented by Ning and Shen (2019) to speedup CNN inference by detecting and exploiting deep reusable computations. This approach can reach near half inference time, without affecting accuracy (loss <0.0005). It is not hardware dependent, so it can be deployed on common CNN accelerators.

Among all layers of a CNN, the convolutional layer is the most compute-intensive one, taking most of the execution time.

The basic idea of this algorithm is to group equal neutron vectors into clusters and use these clusters to compute the operations, then the output is reorganized by splitting the equal vectors grouped in the beginning into the output.

Network compression is a common method that minimizes computation, but rather than compress the net, this technique reduces computation by skipping some operations processing. If required, this technique can also complement a compression method.

Pruning

Pruning is a compression technique that reduces the number of connections of deep neural networks without affecting accuracy Han (2017).

A DNN model can be shorten by removing all connections whose weights are lower than a threshold, converting a dense layer to a sparse one. Synapses (connections) and also neurons can be pruned resulting on a smaller set of operations being processed which reduces inference time.

Once this technique is applied, DNNs size is drastically reduced, which reduces the off-chip Dynamic Random Access Memory (DRAM) access, giving the opportunity to take charge of CPU memory if all weights could be stored on-chip memory, improving inference speed and reducing energy required.

”Since much of the information represented by the weights is in fact redundant and many of the weight values are very close to zero, then we should be able to discard them without significantly affecting the overall performance of the network.” Montantes (2019).

Data Quantization

Data quantization can be used as a compressing technique by reducing the number of bits required to store the weights of the networks. Jetson Nano can take full advantage of this technique since it supports half-precision floating point operations. This makes possible to half the net by representing the weights with only 16 bits instead of 32.

The authors Qin et al. (2018), in their work, apply this technique, adding further compression to a pre-trained floating-point model without re-training it. To evaluate this process a NVidia Jetson TX2 computing system is used.

However, experimental results proved data quantization not to be quite efficient: the inference time onn a quantized model is longer than the original not-quantized trained model.

This technique can speedup computations like matrix multiplications, by avoiding expensive floating point arithmetics and enabling SIMD vectorization by using a compact model. Yet, the de-quantization process creates an overhead during inference time, which adds up more spent time on this process, not benefiting the final results.

The defined de-quantization function that converts input values back to a 32-bit representation on some layers to recover the precision loss is expensive, contributing to 30% to 50% of inference time.

Huffman Coding

A Huffman code is a codification method for lossless data compression. This technique can also be applied to quantized values Polino et al. (2018). The key idea is to encode the weights by computing the frequency of each weight, deriving a table of probabilistic occurred symbols. Finally each weight is represented by a lower-bit representation. The optimal bit length is driven by the commonly weight on the model, saving additional space Han et al. (2016).

2.3 REAL-TIME EVALUATION METRICS

The effectiveness of the autonomous vehicles to be able to process all the information in real time has been an active area of research. When a person is manually driving, multiple factors may affect the traffic flow and even cause hazards, such as human reaction times, distractions and other human errors. To eliminate these human errors and delays, technologies are being developed to replace humans by computer systems.

This vehicle automation may lead to smoother traffic flow and larger traffic flow rates due to the safety among inter-vehicle automatic systems used and the elimination or reduction of human delays and reaction times Ioannou and Chien (1993).

Adaptive Cruise Control (ACC) is a mechanism that aids vehicle on driving with two main tasks: to drive at the maximum possible speed and to avoid collision with eventual vehicles ahead. These tasks are also performed by human drivers, yet this mechanism will often take over the vehicle because it can react in 0.1 seconds, about 10 times faster than a reaction time of a human being Giuffrè et al. (2017).

Self-driving cars are computational heavy by taking full control of the vehicle, which demands multiple requirements to be fulfilled such as memory consumption, processing, latency and throughput. In terms of performance, high computational needs and memory demands are the critical ones, which are the main metrics that will be evaluated on this project. This performance requirements leads to additional challenges while developing and deploying these algorithms on systems with limited computational resources on-board of the vehicles Kouris et al. (2020).

The reaction times to take control of a vehicle during a disengagement was studied by different companies, such as Google and Mercedes-Benz. The performed studies revealed pretty consistent measured reaction times within them, with an average of 0.83 seconds per reaction. These times can provide such an high understandable of how quickly an individual can react upon multiple averse circumstances and even obstacles to avoid collisions. Dixit et al. (2016)

Table 2 shows the reaction times of different age groups. The groups with the shortest and longest response times were selected, 0.74 and 1.17s respectively, for the warning braking distance analysis Li et al. (2019).

Age of driver	Reaction time
18-33	0.74-1.10
34-47	0.75-1.12
48-57	0.77-1.13
58-70	0.78-1.17

Table 2: Reaction time of driver Li et al. (2019)

As human driver reaction times vary on a range of 0.7 and 3 seconds, depending on multiple external variables as the person itself, autonomous driving systems must be low-latency reliable and efficiently take control of the vehicle with the same or better reliability as humans.

Giving the earlier described reaction times of the drivers, building a system that can react within 700 milliseconds or less is pretty acceptable since it outperforms the performance of human reaction's average.

2.4 THE PROBLEM AND ITS CHALLENGES

This work focuses on providing a hands-on hardware and software to optimize deep learning algorithms that can autonomously drive the RC vehicle on a predefined path (Figure 5).

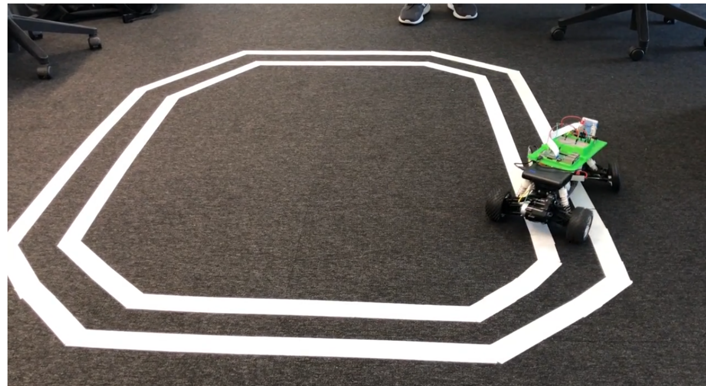


Figure 5: Predefined path

The RC vehicle vehicle was built by modifying an ordinary remote controlled vehicle by gearing it with additional hardware. Several peripherals were installed, such as one microcontroller, multiple sensors, a camera to capture image data of surrounding environment and hardware with computational power to execute the DL algorithms.

Self-driving mode must be activated and available-only in certain conditions, the crucial one is the presence of a road, obviously. The idea of using a real road is inconvenient given the dimensions of the car, aside that testing this small vehicle on a road with several cars passing by is neither practical nor safe. Therefore this testbed will run on a small custom predefined trajectory that mimics a road lane, as illustrated in Figure 5.

Given the dimensions of the ordinary remote control based car, the choice of the hardware platform needs to be kept in mind because this system delivers some constraints, such as dimensions, the weight and of course the costs associated to all the build.

Two embedded systems will be used, comparing two systems that are not equivalent, which both possesses unique hardware characteristics. The process of selecting the proper embedded system for this use-case is also very demanding, if considering all the vast and different types of embedded devices existing in the market.

Also choosing the best hardware equipment is not an immediate and effective task because there is a lot of research implied to, which resides on the chips architecture and what is the impact of its components design like the CPU architecture or the maximum supported bandwidth, and even on the software support and the algorithm technique upon the hardware available.

Therefore providing the maximum performance possible to each system can become extremely complicated due to the amount of combinations existing in one single system with

one common crucial component. For example, despite the same ARM CPU architecture being used, it also has to be considered all the unique specifications of each chip vendors.

The biggest challenge for this project is to deploy the deep learning algorithms on an accelerated neural network inference engine and optimize them so the resource utilization can be tuned by reducing computation and memory consumption in order to the embedded system handle these computational intensive AI workloads. But this optimized process can be achieved by targeting the main unit processors present on the two systems, by enabling multiprocessing on CPU cores and/or targeting the GPU cores.

Taking into account this specific computing platform — embedded systems — the computational power will be naturally limited to this equipment only, since hooking up a dedicated workstation or a conventional desktop on the vehicle is naturally not a option.

The RC vehicle must be able to self-drive on this test environment by deploying, optimizing and running the ML algorithms, specifically deep learning based algorithms on the provided hardware. This algorithms are essentially object detection workloads that could empower a complex and massive number of instructions and operations to be handled by the processing units. The workload comes from the need real-time object detecting from the image data captured by the autonomous car.

Aside the DL workloads that are responsible for taking control of the scaled-vehicle, additional algorithms were also developed to push the vehicle autonomous driving capabilities towards with additional features that are a must when driving on an environment with traffic signs.

Autonomous vehicles must promptly and efficiently obey the most diverse traffic signs that eventually will occur during the driving on a certain path. For this project, an extensive work was also developed that aimed to develop, deploy and evaluate DL algorithms capable to detect traffic signs. Besides the vehicle upon traffic signs reaction must also react towards the information processed.

Finally there is a need to improve the performance of these algorithms so the car can process an higher amount of information per unit of time to be able to speed up and smoothly drive even on tight bends. Also with an improved performance the algorithm can be scaled up, and complexity to the CNNs architecture can be added. An improved convolutional neural network can extend the vehicle decisions logic and intelligence, like, the possibility to read and interpret traffic signs.

While operating under any convolutional neural networks, two intensive and dependent workloads are defined: the training and the inference process. For this testbed, training phase can be performed on a remote workstation, since the real-time self-driving mode can be achieved with inference only. The training phase can be processed before the setup and deployment of all earlier described components, so it can be handled on a high-performance computing platform, relieving the embedded systems of training process workload. The

scope of this thesis work is also focused on the inference phase and its optimizations, so only inference challenges are targeted to the chosen platforms.

INFERENCE CHALLENGES

A deep neural network comprises an architecture specified by multiple layers, and for most use-cases, as the model gets deeper, the inference accuracy increases. This leads up to high computational demands that embedded systems could not pace, unlike powerful equipment whereas a desktop or datacenter systems can be used to process these tasks.

Given the urge to deploy the deep learning applications in embedded platforms, there is a need to make feasible the support of the workloads alike to predict and produce intelligent decisions. In addition, these devices have to cope with additional constraints like low-latency execution and power consumption.

Advanced computing libraries for embedded devices are missing or require optimizations due to the broad range of existing architectures and diversity of programming tool-chains Irida Labs and Silexica (2018). Hence, most of the optimized libraries aim to improve training performance, however inference process is lacking of these advanced optimized tools.

This is an issue since the already existing inference engine tools are architecture exclusive such the ones supporting NVidia GPUs only or a small range of CPUs, specifically ARM Cortex.

2.4.1 *Proposed Approach*

To accomplish the goals of this dissertation a defined system architecture is presented (Figure 6), where three main concepts need to be studied and mastered: (i) **hardware accelerations**; (ii) **accelerated NN inference engines**; and (iii) **network optimizations**, so DNN inference optimizations can be achieved, increasing the number of frames being processed by the real-time object detection system.

Next are described the main core components that have to be analysed in order to improve the system performance of the DL algorithms.

Hardware Accelerations / Embedded Systems

To target embedded optimizations, two embedded systems were selected: Raspberry Pi 4 and NVidia Jetson Nano.

These both platforms have ARM Cortex-A built-in processors that provides several advanced computational features regarding to accelerations provided by HW-parallel implementations present in the CPU.

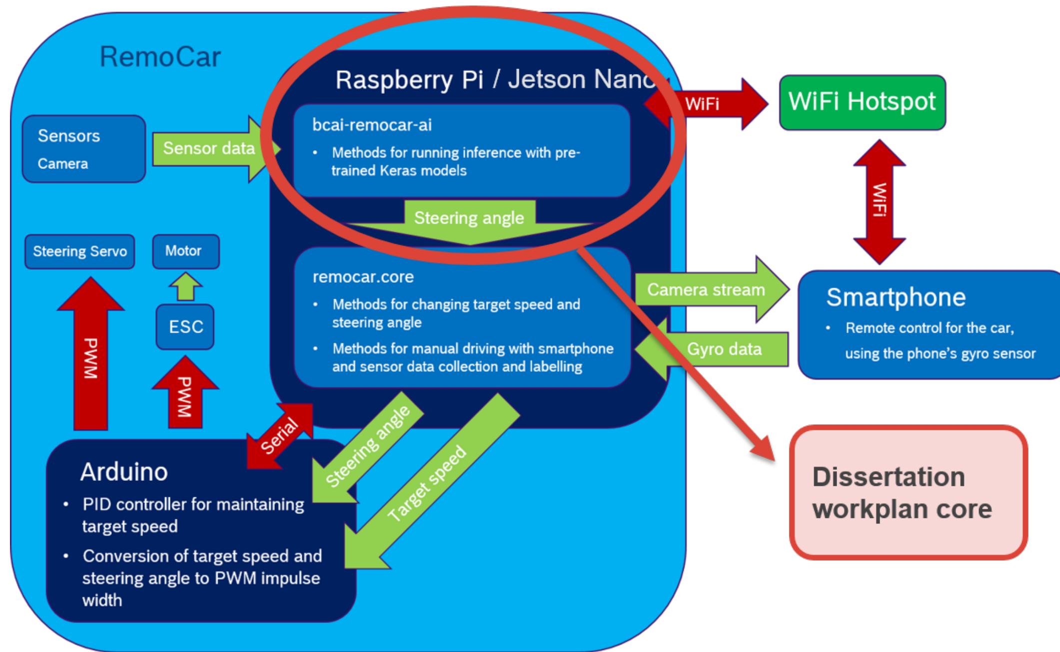


Figure 6: System architecture

The NVidia system-on-a-chip also provides a dedicated GPU to improve the chip computational power, possessing two processing units.

Embedded devices typically require in-depth knowledge of its hardware architecture and programming interfaces that are quite some, in opposition with desktop application development. The packages support for these devices are usually very limited, an evident problem seen on modern embedded platforms that hold on a heterogeneous set of processors like CPUs, GPUs and others. Irida Labs and Silexica (2018)

Thus the embedded systems must be very well understood to maximize and enable all the advanced parallel and computing features provided by the hardware. This advanced instructions should be powering the performance matter so developing algorithms or using frameworks that are aware-of these HW capabilities are quite important.

Inference engines

It is an urgent need to analyse the inference engines in the market so the accelerated ones can be discovered, such the ones that are able to exploit the advanced computing instructions and HW capabilities, such as the parallel instructions use or even the ability to enable GPU cores.

All the algorithms must be benchmarked so the program trace can be evaluated, understanding the algorithm limitations and what bottlenecks can be avoided.

CNN optimizations

The proposed hardware platforms could not have enough computational power given the use-cases regarding to the self-driving area. However the CNN models and algorithms used throughout the project can be enhanced and optimized using multiple hardware advanced optimizations available on inference engines and using techniques to optimize the networks.

By applying some optimizations to the CNN, rather on the model itself or enabling hardware features capable to tune this kinds of neural networks will lead to achieve the desired goal to improve the performance of the DL algorithms.

Figure 7, displays the concept map of this dissertation workplan, where the main concepts and technologies being used are relationally linked.

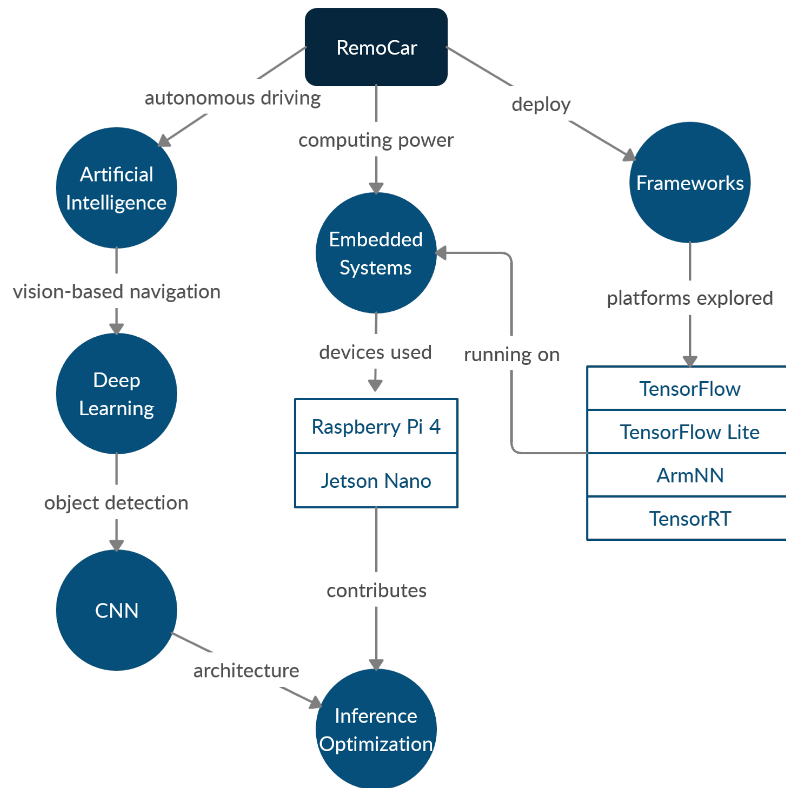


Figure 7: Concept Map

The key concepts are displayed in boxes or circles, which are connected with labeled arrows. These relationships defined by the links can articulate the concepts in a "cause", "requires", "such as" or "contributes to" effect.

3

ML INFERENCE ON EMBEDDED DEVICES

3.1 THE TESTBED ENVIRONMENT

3.1.1 *The RC Vehicle*

This section presents all the required hardware to build the desired RC vehicle that acts as a testbed for this project. A brief explanation of how the vehicle autonomously works is given, as well as the process to manually drive it and to collect the training data, which is mandatory to enable the ability of the autonomous driving.

The RemoCar is a 1:10 scaled vehicle and it is composed by five main electronic components (Figure 8): a camera, a single-board computer (the Raspberry Pi in this image), a microcontroller (Arduino), a servo and an Electronic Speed Control (ESC).

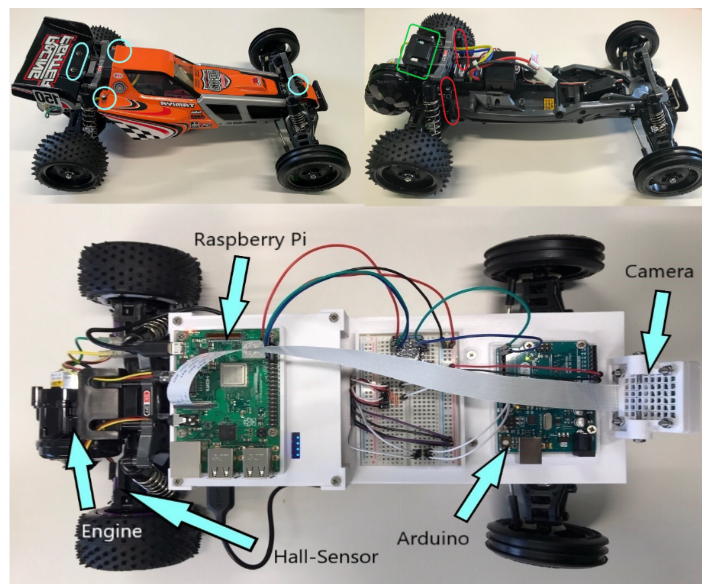


Figure 8: RemoCar essential components

These components were provided by Bosch together with the vehicle itself, which came all disassembled. The testbed was then built by assembling the several parts of the vehicle chassis

(Figure 9). The electronic circuit (breadboard) that is between the single-board computer and the microcontroller was also wired from scratch.

Figure 9: RemoCar chassis parts

The camera allows the vehicle to have a visual perception of the environment, allowing to recognize the track, with about 4 or 5 frames per second, originally.

These images are transmitted to the single-board computer — which can be either a Raspberry Pi or a Jetson Nano — where the pre-trained neural network will run on. These devices constitute the computational power of the RemoCar.

The Arduino UNO is an open-source microcontroller board, which takes control over the steering and the engine of the vehicle. It uses an electrical signal to transmit the output received from the single-board computer to the servo and to the ESC.

The servo is a rotatory actuator that produces rotatory motion which allows the wheels to be steered. The ESC is an electronic circuit that controls and regulates the speed of the vehicle.

The main goal of the RC vehicle is to autonomously drive on a track by following the lines of the path, as shown in Figure 10.

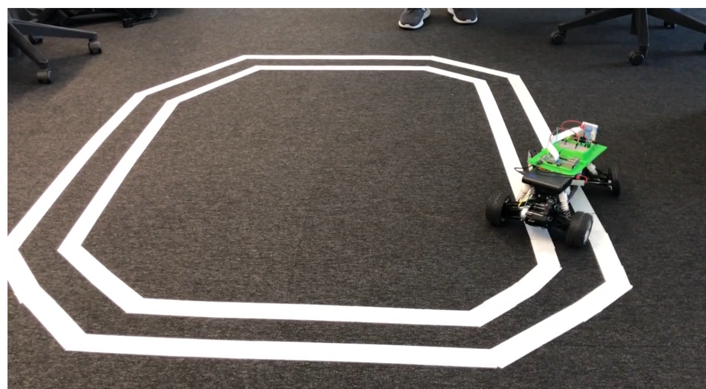


Figure 10: RemoCar following the track

The ability to follow the track is given by a neural network. The neural network provided can recognize a path, perceive if the lane roads are visible, understand the orientation of them and predict a steering angle that allows to continuously adjust the wheels according to the path.

Initially, the RC vehicle (Figure 11) cannot autonomously drive on a path because the neural network does not know what is a path and what is not, does not know what is a steering angle and how to map a certain line orientation to a steering angle.

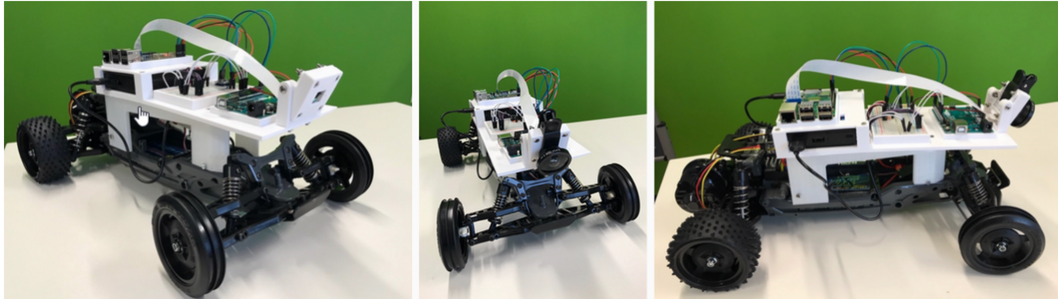


Figure 11: Assembled RC vehicle

Manually driving the vehicle

Like any neural network, the ones used in this testbed needs training data. In this use-case, this data is collected while manually driving the vehicle using a smartphone as the vehicle remote controller. The single-board computer that runs the neural network also hosts a web server that can be accessed by WLAN.

After establishing an internet connection on the same network as the single-board computer is connected to, the car can be manually driven by tilting the smartphone.

While the smartphone is being tilted horizontally, the single-board computer extracts the gyro data from it and sends the recalculated data via serial communication to the Arduino.

During manual driving, the video-camera is activated, recording all the driving and storing the frames into storage. For each frame recorded, the gyro data of the smartphone is also linked to the image and stored in a CSV file.

After reaching a predefined number of pictures recorded on training data, the car stops automatically, means that it already collected a sufficient amount of frames (training data) to be used to train the model.

After collecting the training data, each image needs to be manually classified, whether the track is visible or not, as a prerequisite to train the network. This classification requires an amount of time since each recorded image needs to be checked manually and needs to be moved to a specific folder whether the track is visible or not.

Next, the classified training data can be processed and the neural network can be trained. The Raspberry Pi cannot train the network because the computational power is not sufficient,

so this operation may be executed on an external computer or on the Jetson Nano. All the algorithms and the environment to properly train the neural network was provided by Bosch Car Multimédia, S.A.

The trained model is then exported, generating the required file so the network can later properly identify the track and predict steering angles. The RC vehicle can now be autonomously driven using this generated trained model.

Before the autonomous driving mode can be enabled, the vehicle must the placed into the track. When the RC vehicle vehicle is powered-on on this mode, the neural network compares the current seen image with the pictures from the training data and based on the learned process it predicts a steering angle if the lane is visible.

If a steering angle is predicted, the single-board computer transmits it to Arduino via serial communication, which applies a constant speed to the ESC and transmits back the angle to the servo.

The whole process from comparing the current seen image to sending the information to the servo and ESC is repeated indefinitely, checking every frame. The model predicts whether the lane is visible, and if it is true, an appropriate steering angle is predicted. In case the lane is no visible in sight, the vehicle turns fully to the last known direction. The speed is always constant.

3.1.2 *The Arduino Microcontroller*

Arduino is an open-source platform to build electronics projects. It consists of both a physical programmable circuit board, referred as a microcontroller, and an Integrated Development Environment (IDE) that runs on the host computer, used to write and upload the developed code to the physical board.

Unlike the majority of programmable circuit boards, the Arduino does not need a programmer — a hardware component — to flash the code onto the board. Only a USB cable, connecting the host computer to the microcontroller, is required to perform this operation.

This platform was designed for anyone interested in creating all kinds of projects with interactive objects and environments, controlling electronics. Arduino can interact with a wide-range of electronics like motors, cameras, controllers, GPS units, networks and even smartphones.

Its flexibility combined with the free Arduino software, the vast community contributing to a huge variety of Arduino-based projects, the easiness to learn both hardware and software that presents a syntax similar to C++ and the low-cost hardware boards, contributes with strong motivations to use this type of board on this testbed.

Cross-platform is also an advantage of this ecosystem, which means it is possible to develop and flash code from Windows, Macintosh OSX and also Linux operating systems.

There are multiple Arduino boards, on which its hardware components differ from one to another. For this use-case, it will be used the basic Arduino UNO because it is one of the cheapest boards and it is not necessary a powerful one with additional components. This device is displayed in Figure 12. Some boards may look different among the different Arduino microcontrollers, however the majority of them carry the components numbered on the image.

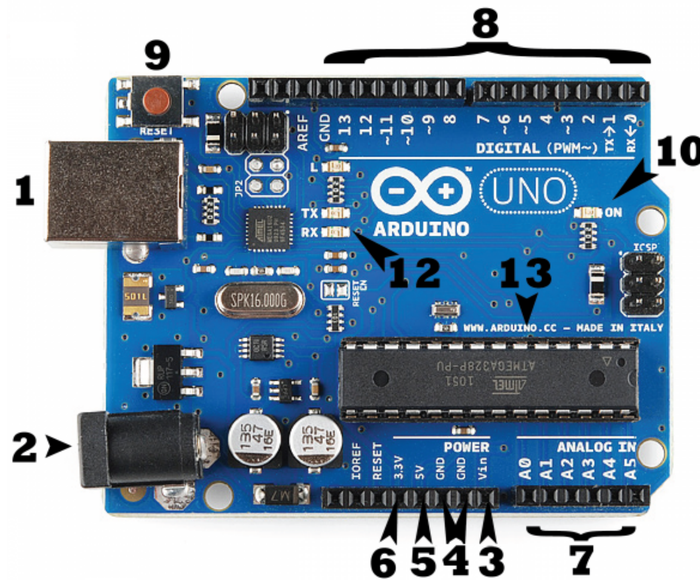


Figure 12: Arduino UNO board

Every microcontroller board needs to be connected to a power source, so this Arduino board can be powered up using a USB cable (**1**), using wall power supply by connecting the DC connector (**2**) or directly using the power pins (**3**). The USB connection is also the way to flash the code developed on the IDE onto the microcontroller memory.

The board also offers a set of pins where wires are connected to build a circuit. There are multiple kinds of pins that fulfills different purposes. The Vin pin (**3**) is designed to either power external devices or to power in the board from an external power source. The GND (Ground) pins (**4**) are meant to ground the circuit built. The 5V (**5**) and 3.3V (**6**) pins supplies 5 volts of power and 3.3 volts of power, respectively, useful to power up external components. The area of Analog In pins (**7**) can read signals from analog sensors, like a temperature sensor, and convert it into a digital value readable by the board. Digital pins (**8**) can be used for both input and output, such to recognize when a button was pressed on or to light up an LED.

A reset button (**9**) is also available, pushing it will restart the code being executed. The LED ON (**10**) lights up whenever the Arduino is powered to a power source. TX (Transmit) and

RX (Receive) LEDs (**12**) indicates whenever the Arduino is transmitting or receiving data on serial communication.

The Integrated Circuit (IC) (**13**) is the microcontroller device responsible for the code execution.

This device is connected to the ESC and to the servo where it sends instructions to set speed and wheels direction to the vehicle. These instructions are received through a third connection, the single-board computer, which sends the computed steering angle and the desired acceleration. All these connections were performed on a breadboard, which facilitated the wiring connection between all components, as presented in Figure 13.

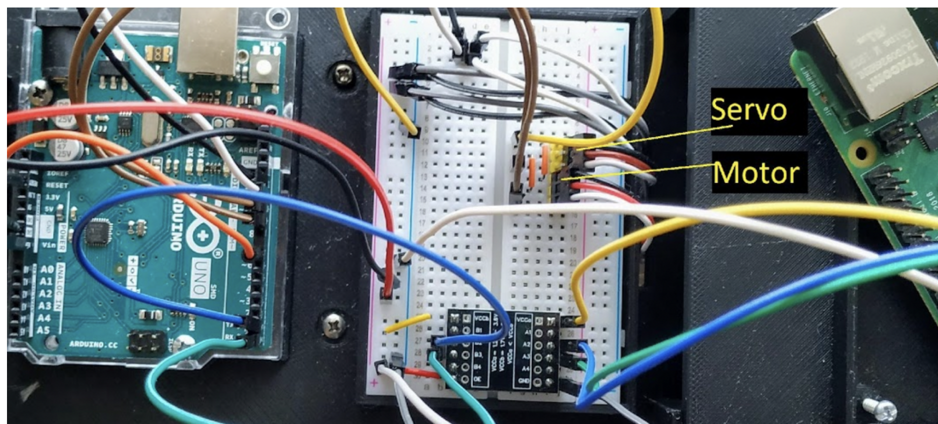


Figure 13: Wiring connection. Breadboard (center), Arduino (left), board-computer (right)

3.1.3 *The Raspberry Pi and Jetson Nano Embedded Systems*

Two different single-board computers were selected to deliver computational power to this testbed environment. The main role of this platform is to execute the neural networks, which allows the autonomous driving itself.

The chosen computers to run this ML workload are a Raspberry Pi 4 and a NVidia Jetson Nano, both illustrated in Figure 14. The goal is to test and install each computer at a time on the RC vehicle, deploy the neural networks on the devices, improve the performance on each computer and evaluate both devices to choose the most efficient one.

These two embedded systems have distinct properties: the Rapsberry Pi has a better multicore ARM device but the Jetson Nano has a GPU accelerator. The cheaper Raspberry Pi 4 version costs only $35, whereas the Jetson Nano starting prices are around the $99. Each computer behaves differently and the type of workload being executed also influences its performance. It is important to evaluate both embedded systems and choose the right one for this use-case. Choosing the right single-board computer will allow the vehicle to react faster and process more data in real-time, while driving autonomously.

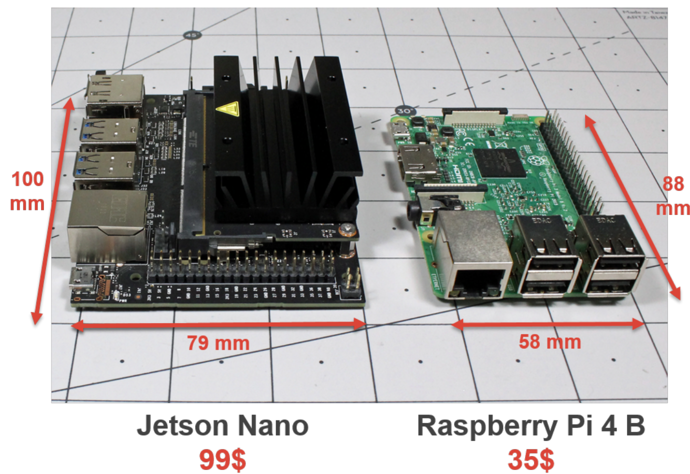


Figure 14: Target embedded systems

Raspberry Pi 4

Raspberry Pi 4 is a tiny single-board computer. It is faster and more functional that any of its predecessors versions, launched at the same price of the previous one (Raspberry Pi 3).

This fourth version of the Raspberry Pi series empowers a 64-bit ARM Cortex-A72 quad core processor on 1.5 GHz and 4 GiB of Random Access Memory (RAM). This pipelined processing unit has 5-wide out-of-order, speculative issue 3-way superscalar execution pipeline. Each core supports the 128-bit NEON SIMD extension and has a VFPv4 — a Floating-point Unit (FPU) coprocessor extension, onboard equipped with 64-bit FPU register as standard, half-precision support as storage format. Table 3 presents a summary of the hardware characteristics.

CPU	Cortex-A72
Cores	4
GPU	Broadcom VideoCore VI (32-bit)
L1 cache (KiB)	48 (instructions) + 32 (data)
L2 cache (MiB)	4
Clock frequency	1.5 GHz
SMT	None
Out-of-order	5-wide dispatch
RAM (GiB)	4

Table 3: Raspberry Pi 4 hardware specifications

A second processing unit is also available on this embedded system: a GPU. The Broadcom VideoCore VI graphic card installed is a low-power mobile multimedia processor but it is for video rendering purposes only because the Raspberry Pi provides two 4K output video ports

that requires high-speed video processing. Although two processing units are available on this single-board computer, only the CPU can be exploit to process the DL algorithms.

Jetson Nano

Jetson Nano is a low-power computer-board produced by NVidia, designed to accelerate machine learning algorithms.

This platform also contains a quad core ARM Cortex CPU. A predecessor of the CPU used on Raspberry Pi 4 is used, the 64-bit ARM Cortex-A57 chipset with 1.43 GHz of clock frequency. The processor provides the same advanced instructions as the ARM Cortex-A72 described earlier. Table 4 summarizes the characteristics of this embedded devices.

CPU	Cortex-A57
Cores	4
GPU	NVidia Maxwell
L1 cache (KiB)	48 (instructions) + 32 (data)
L2 cache (MiB)	2
Clock frequency	1.43 GHz
SMT	None
Out-of-order	3-wide dispatch
RAM (GiB)	4

Table 4: Jetson Nano hardware specifications

The main difference between this Cortex-A57 and the Cortex-A72 earlier mentioned is the clock frequency and the size of L2 cache. Jetson Nano CPU clock frequency is a little lower than the Raspberry Pi, providing 1.43 GHz against the 1.5 GHz yielded by the Raspberry Pi CPU.

CPU also delivers less advantages regarding to the advanced instructions such as the width of out-of-order execution dispatches.

The biggest advantage of this NVidia embedded device is the second processing unit present on the chip, a GPU, which can the exploited to run explicit programmable algorithms.

Its GPU carries 128 Maxwell generation cores with a clock frequency of 921 MHz, giving a throughput rate of approximately 472 Giga Floating-point Operations Per Second (GFLOPS). It also has support to FP16 operations, which can theoretically double its throughput. Details of this second processing unit is presented on Table 5.

Jetson Nano also delivers multiple power modes that can be easily configured by selecting one of two predefined power configurations available. Thus the end-user can easily tune the device to adapt the performance and power needs to the desired needs. Sometimes this operation can find an ideal configuration that best suits the type of algorithms being deployed.

GPU (chip)	Tegra X1 (GM20B)
CUDA-cores	128
Architecture	Maxwell
Clock frequency	921 MHz
GFLOPS	472
Half-precision support	Yes *(not tested yet!)*

Table 5: Jetson Nano GPU specifications

To get all the power out of the Jetson Nano, it is necessary to select the MaxN (10 watts) mode. This disables all the restrictions on the hardware capabilities, delivering all the power the hardware can get.

The second mode – 5W (5 watts) – optimizes power efficiency and restrict the modules to a predefined configuration by capping the memory, CPU, and GPU clock frequencies, and the number of online cores at pre-qualified values NVidia (2020).

Table 6 depicts the supported power modes of Jetson Nano, as well as the differences between.

NVPModel clock configuration		
Property	**Jetson Nano**	
	MAXN	**5W**
Power Budget	10 watts	5 watts
Mode ID	0	1
Online CPU	4	2
CPU Max Clock Frequency (MHz)	1479	918
GPU TPC	1	1
GPU Max Clock Frequency (MHz)	921.6	640
Memory Max Clock Frequency (MHz)	1600	1600

Table 6: Supported power modes of Jetson Nano

3.1.4 *The ML Inference to Follow a Path*

The algorithm that is responsible to follow a path uses a specific neural network that is able to process images and predict to the steering angle and if the lane of the path is visible.

The neural network provided is a CNN composed by twelve hidden layers (Figure 15). The input layer is designed to parse an image of a fixed pre-defined resolution. On the final layers, the network diverges onto two different ramifications and ends with two different layers: a regression and a classification layer. This means that from each processed image, the neural network will predict two values: the steering angle and the visibility of the road lane.

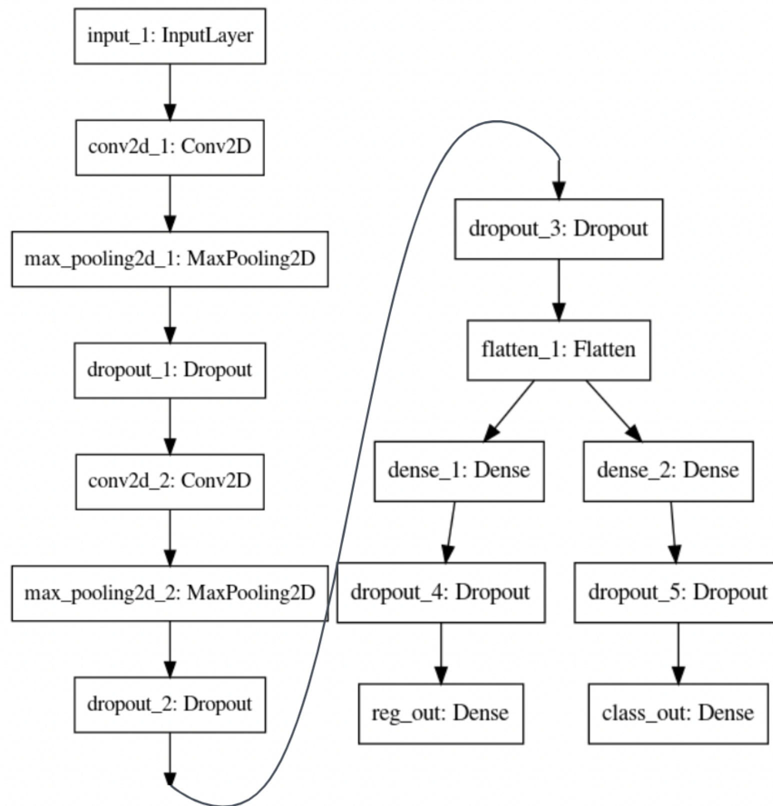


Figure 15: CNN architecture

During the autonomous driving, the camera is enabled and starts to record each frame, which is sent to the model that was previously trained. The CNN processes the input image and by iterating it over the multiple hidden layers, it tries to find the line on the track (Figure 16). While analysing the image two values are being generated.

The output of the classification layer represents the probability the model had found the track line. On the algorithm a predefined threshold is used, and if this predicted value is less than 95% it is interpreted as if no track was found ahead of the vehicle.

Figure 16: Input image example

Another value is predicted by the model through the regression layer. This value represents an angle between -45º and 45º. This is the steering angle that should be applied on the vehicle's wheels so it can properly follow the path.

All the inference phase described earlier is only possible if the CNN had been trained earlier. If not, the neural network will not have the weights of each node adjusted to find a track on a image and will not be able to predict reliable information.

This neural network will only provide intelligent information if there is knowledge on a training dataset that can be learned from it. On this use-case, the training data represents the visual confirmation of the path, the aspect of the lane and the steering angle that needs to be applied on each point of the path.

To create a training dataset, the vehicle must be manually driven and essential data must be logged such as image recording of the track and the steering angles the user applied during the route, as mentioned eariler.

When enabling manual driving mode, the single-board computer that is hosting the web-server receives the desired orientation of the wheels, and sends this data to the microcontroller, which consequently instructs the servo to adjust the wheels to the desired angle. In parallel, this data is being recorded to log-files, by linking each frame recorded to the gyroscope data collected at this time. After a certain amount of images had been saved, the vehicle stops, ending the training data collecting phase.

A manual processing of the collected data must be executed then. Thousands of frames and the respective steering angles were recorded, yet, the manual driving is not perfect and sometimes the vehicle can get off the track. During those moments the recorded frames do not present any track, and these types of images must be manually moved to a specific folder which represents images that does not contains the track. This filtering will separate the images which contains the lane of the track from those that do not. Now, the neural network can be properly trained, it will able to recognize the track by analysing the similar patterns present on the frames of the the visible lane folder. For each processed lane the corresponding steering angle is also learned so the network can predict this value for similar images.

After the CNN had been trained, the trained model will be exported, which is the model that must be imported by the inference libraries or frameworks. With this imported model on a engine, whenever it receives an input image, it outputs two predicted values.

3.2 SETUP AND SOFTWARE INSTALLATION ON THE RC VEHICLE

3.2.1 *Setup the Environment*

This testbed requires specific algorithms and frameworks to be executed and deployed on specific embedded devices. This process led to the initial effort of this project that was carried

by the setup of the two different environments — the Raspberry Pi 4 and the Jetson Nano — while installing and configuring all required packages, libraries and specific versions to run the multiple DL frameworks and libraries tested: TensorFlow, TensorFlow Lite, Arm NN and TensorRT.

Raspberry Pi 4

A full software setup of the Raspberry Pi was performed from scratch. This included the installation of the operating system — Raspbian 32-bit OS — onto the platform. Also, a setup of the 64-bit OS version was performed to evaluate the execution of the algorithms on this specific version, during a debugging phase. Results showed this OS variant did not impacted the performance of the algorithms and did not biased the RC vehicle driving performance. This evaluation was achieved because the Raspberry Pi 4's CPU has 64-bit support. More details of this operation is given in Section 3.3.3.

To run the DL algorithms on this platform, Tensorflow, TensorFlow Lite, Arm NN and PyArmNN were installed on it. All these installations should had been quite trivial but the fact that all setup was done into embedded systems running on an ARM CPU architecture, slowed and complicated this process. The installation of the frameworks and specially its dependencies was time consuming, since several incompatibilities were found on an ARMv8 system architecture, rather that on a AARCH64 system.

Additionally, the machine learning packages and the compatibility among the multiple versions of its dependencies are constantly being modified and upgraded, which adds up a new complexity to perform a proper setup on a specific moment, since new releases sometimes brings incompatibilities among AI libraries and packages.

This leads to a highly volatile environment, where a stressful setup for a given environment at one time can become quite trivial after a period of time. That was also witnessed on this project. A quite expensive setup was performed on the initial phase of this project because the support for the platforms was not the ideal for the required libraries used. An active monitoring on the versions of the frameworks and their dependencies was performed, which showed an increase support of the tools used for these embedded devices, later in time, allowing a light setup.

The initial deadlock on the setup was due to major incompatibilities of some libraries with the Raspberry Pi operating system and ARM architecture. This led to quite some spent time on founding out the right compatible versions with ARM architecture and finding the right combination of package versions that were compatible with each other.

Initially, it was installed TensorFlow version 1.14 on Raspberry Pi 4, the latest version officially available on the initial phase of this project development. These specific versions was also found out not to load properly specific neural network layers. The limitation was

resolved on TensorFlow 2, but once again it was not officially supported by ARM on that moment. Later, at the end of this dissertation work, a support to ARM was released.

Jetson Nano

A full setup was also performed on the Jetson Nano device. NVidia, the company supporting this platform, provided an automatized environment, which streamlined the installation of all required software to run the DL algorithms on its platform and also taking advantage of GPU-cores.

Multiple evaluations and an extensive comparison against the predicted data on the Raspberry Pi 4, showed critical anomalies on the installed TensorRT library. The TensorRT version 5 initially installed could not properly load multiple layers of the different neural networks used. The library behaved like the predicted data was being properly generated but in fact it was misleading information because TensorRT 5 output accuracy decreased comparing against the Raspberry Pi 4. This critical internal bug was later fixed with the release of TensorRT 6. More details of this problem is presented on Section 3.4.3.

3.2.2 *The ML Inference to Detect Traffic Signs*

The original algorithm, which is capable to follow a path, was successfully deployed and its performance was tuned and improved. The evaluation results showed both the Raspberry Pi 4 and the Jetson Nano could easily process this workload without any limitations or drawbacks.

As both embedded systems could easily process an higher amount of frames per second while processing the images and predicting the steering angle, it was necessary to develop an additional and computational heavier DL workload to perform a deeper analysis of both embedded devices performance and limitations.

Therefore, an additional algorithm, which is able to detect traffic signs was developed. This algorithm, like the one responsible to drive the vehicle, also uses a neural network to process an input image and predict information relative to the existence of traffic signs on the road.

For this use-case, a neural network was used, FgSegNet, which was introduced on an earlier chapter, to detect traffic signs in an input image.

FgSegNet takes an input image, processes the data and proceeds to the segmentation of certain regions that composes a traffic sign. After the image had been processed by all layers, the last layer generates an output image, where the regions with traffic signs are segmented.

Training

Initially, this network is not intelligent and cannot recognize a traffic sign. Without a previous training, the neural network has no ability to proceed to foreground segmentation, thus not yielding any segmented object.

All DL algorithms that uses trained models are composed by two phases — training and inference. Inference phase, on this use-case, is the process of predict any traffic sign present on an image. Training phase is when the network is trained by analysing patterns on a given training data.

The training data must contain the attributes and the targets. The algorithm will analyse the patterns of all attributes and will learn how to map them into the targets, generating an ML model that maps (predicts) different attributes into desired targets, following the same patterns of training data.

In this case, the attributes are images of an environment full of traffic signs and the targets are images with the segmentation of traffic signs, as presented in Figure 17.

Figure 17: Segmentation of traffic signs. Attribute (left), target (right)

The importance of Cityscapes dataset comes in this training phase because it provides a large amount of tested data, acting as the ground truth for training the DL model. Cityscapes delivers an extensive training data, which contains the attributes and targets needed to successfully train the FgSegNet.

Cityscapes does not isolate the traffic signs objects and presents the segmentation of multiple types of objects, as previously presented in Figure 2.

Thus, a script to process data-images from a dataset was developed, in order to analyse all the annotations of segmented areas and extract only the ones in yellow (which the Cityscapes identifies as traffic signs objects). Figure 18 displays the original segmentation of multiple objects present on the original dataset (on left) and the isolation of traffic signs (on right).

The script processed all the annotations of a large number of segmented images to generate correspondent images with the annotation of the traffic signs only. This process generated a large isolated traffic signs dataset containing a considerable amount of image pairs defining the attributes and its target (Figure 17). FgSegNet was trained with this isolated dataset.

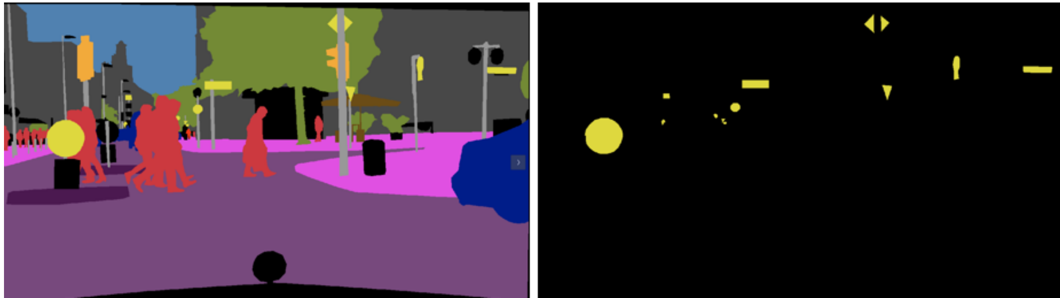


Figure 18: Isolation of traffic signs segmented areas

Prototype Constraints

With the arrival of the second model — traffic signs recognition — there is a necessity to resort to multiprocessing techniques to properly execute both models in parallel.

Therefore an architecture was defined (Figure 19) that follows a non-sequential application execution, which is efficient to compute the predictions of both model instances. The prototype design also manages to open two concurrent communication channels to directly send instructions to the Arduino, which controls all motor functions of the RC vehicle.

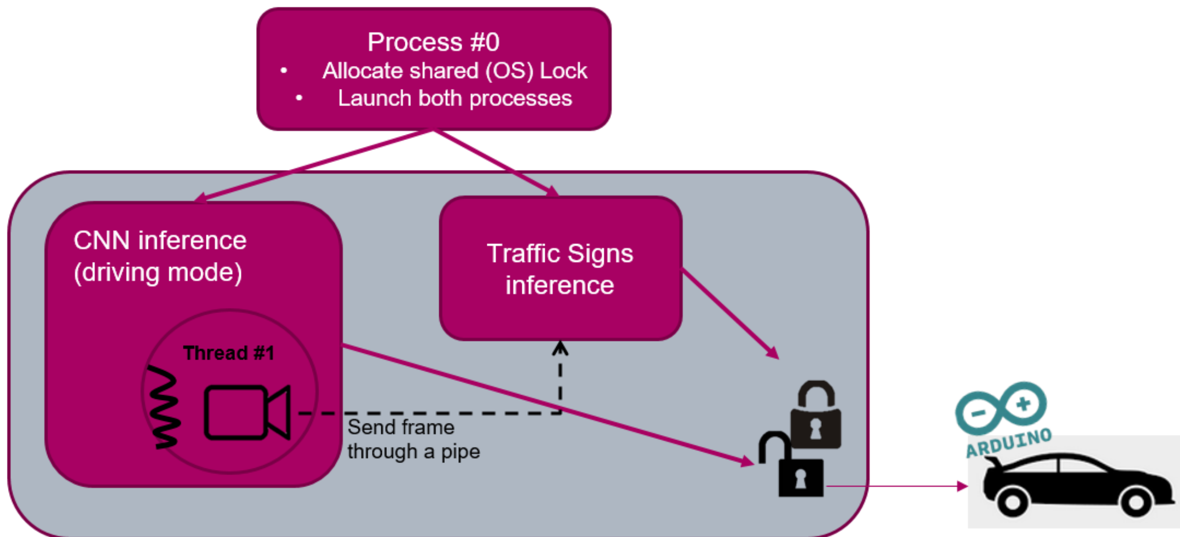


Figure 19: Parallel architecture to run both DL workloads

Two deep learning workloads need to be executed: the original CNN model, which is responsible to predict the steering angle, and the traffic signs recognition model.

To independently run these workloads, two processes are created on runtime. Each process is responsible for one model allocation and execution.

The process responsible for the CNN inference follows an architecture that is detailed on Section 3.2.2. Summarizing, this algorithm that runs on a single process, creates a thread that is responsible to capture real-time images, while the vehicle is driving. Each image is sent

to the main thread to be evaluated by the CNN model, predicting the desire steering angle and the road lane visibility. After this phase, the steering angle and a defined constant speed value is sent to the Arduino, so the vehicle can speed up and steer its wheels.

Alongside this architecture, lies the second process that supervises the traffic signs algorithm. As mentioned before, this algorithm, similarly to the CNN inference, also predicts information by processing frames in real-time. Unlike the CNN inference process, on traffic signs inference process, there is no need for a thread that deals with image capture to be created.

As both processes should be simultaneously executed, in parallel, and the first process is already capturing images, there is no need for the second process to also perform the same operation of capturing image data. If image recording was duplicated on the second process, a lack of performance would occur because the system would be unnecessarily performing the same operation twice.

To avoid the creation of a similar thread inside the traffic signs process, a communication channel was established between the two algorithms. For each frame captured, in real-time, by the correspondent thread, an additional task is now performed before the images be accessed by the CNN model. The thread starts by establishing a communication channel with the other process via message passing, and each frame captured is now sent to the address space of the other process — traffic signs inference algorithm.

On the other hand, traffic signs algorithm receives the images on the output of the communication channel, sent by the previous process. Each frame received is redirected to the FgSegNet model created, which predicts the masks of potential traffic signs present on the image, as presented in Figure 20.

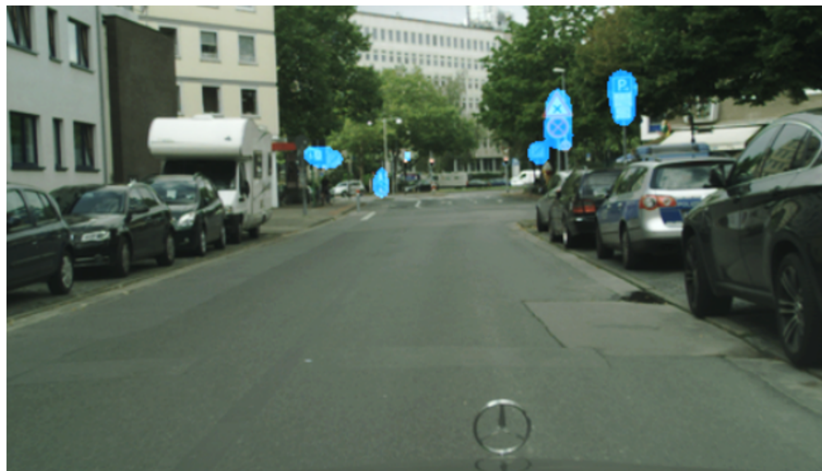


Figure 20: Output of traffic signs inference: masks of predicted traffic signs.

Consequently, a postprocessing operation is performed on the output of each prediction, which may contain a set of traffic signs masks. To easily analyse the predicted image, all

pixels that does not contain a traffic sign are discarded, keeping only the regions of interest (ROIs), which define the sets of predicted traffic signs masks.

Regions of interest (ROIs)

The Regions of interest (ROIs) computed are defined only by the masks of the predicted traffic signs, as showed in Figure 21, discarding all the pixels outside the predicted area.

Figure 21: ROIs of the predicted masks from Figure 20

Nevertheless, there are some constraints to the RC vehicle motion if it only reacts upon the regions detected. Originally, the predicted masks does not indicate any proximity sense of each segmented traffic sign, or if the sign is near enough of the car to make it stop, like a stop sign. Therefore, each detected ROI was post-processed in order to find the total dimension of each region.

Figure 22 displays the analysis of the previous computed regions, shown in Figure 21. This refinement can provide to the algorithm a sense of dimension of each predicted traffic sign mask.

```
 ex_mask/sinal2.png  has elements:  5
99.0
8.0
189.5
551.0
389.0
```

Figure 22: Dimension of the ROIs computed in Figure 21

Another advantage of detecting the dimension of each region is the possibility to discard any predicted mask that shows poor results. An example of these types of results are presented in Figure 23. The predicted masks can correctly identify the traffic signs cluster present on the

given input image. However, there are regions where the masks are too small, and processing the ROI dimension indicates the traffic signs are too far to be reacted to. Another visible case, is a small region, which does not contains any traffic sign. These small predicted clusters will be discarded by the algorithm because it may contain wrong predicted masks or may represent traffic signs too far from the vehicle. On both cases, the RC vehicle will ignore this predicted information.

Figure 23: ROIs of predicted images with poor results

Several experimental evaluations were performed to find an optimal cluster pixel dimension, on which the RC vehicle should only react to. Given the experiments, 600 pixels is the minimum dimension of each ROI, the vehicle should react. Any cluster dimension smaller than 600 pixels will be ignored, which means these masks may be relative to far detected traffic signs or wrong predicted masks.

As described earlier, Cityscapes can recognize a wide-range of European traffic signs, however, the predicted information does not identify each traffic sign type. Therefore, since the DL model will not have the ability ti distinguish between the multiple types of traffic signs, each predicted mask will be interpreted as a stop sign, for this use-case. This is because intends to explore the performance of DL algorithms and not to enhance the reaction abilities to traffic signs of the vehicle.

Traffic signs inference algorithm finishes with a communication phase. After traffic signs had been predicted and ROIs had been processed, an action order — *stop* or *continue* — is directly sent to the microcontroller.

Each traffic signs cluster will be analysed and if there is at least one region greater than 600 pixels, means a traffic sign is near. Upon this information, a direct stop order is sent to the Arduino, ordering the vehicle to stop during 5 seconds upon the stop sign ahead. If none

regions are detected or if are smaller than 600 pixels, the vehicle will ignore the information and will keep up following the lane.

As illustrated previously in Figure 19, the both processes that are running in parallel will eventually send instructions to the microcontroller. However, this concurrent operation may lead to the corruption of the messages in case they are sent at the same time. To prevent any message overlap, the communication channel is protected with an Operating System (OS) lock object.

The allocated primitive lock is shared among both processes, and will guarantee the messages are atomically sent. This communication synchronization is established because serial communication between Raspberry Pi or Jetson and the Arduino cannot protect each entire message structure on multiple concurrent sent requests. Leading to the information sent be overlapped consequently by the multiple processes, destroying the original data.

A primitive lock could be in one of two states — locked or unlocked. The first process to send an instruction to the microcontroller will lock the synchronization object, preventing any other eventual send request by another process while the previous message is still being sent. After a message had been sent to the Arduino, the primitive is unlocked, allowing any potential pendent request to be sent to the messages queue.

3.2.3 *Performance Evaluation*

Develop efficient algorithms or fine-tune pre-existing ones requires the usage of feasible evaluation tools or techniques able to properly analyse and test them. Since this project requires the development of multiple algorithms and an extensive evaluation of the different developed versions, there is a necessity to develop a tool orientd to this use-case to aid on this process, automating this task.

The critical metrics of the real-time algorithms that must be analysed are inference times and memory consumption. These metrics will aid on choosing the efficient DL inference library or framework for a certain embedded device.

Manually executing and collecting each metrics of the multiple algorithms running on different frameworks, would be time consuming, requiring an automatic performance evaluation. To easily retrieve the needed metrics, per inference execution, an evaluation script was developed.

Figure 24 represents the architecture of the developed tool. The main shell-script calls an OS tool named VMstat, which retrieves the amount of memory the system is using, before executing the DL workload.

VMstat is a computer system monitoring tool that analyses multiple system metrics, such as, memory, processes, interrupts, paging and block I/O. The process of collecting the system

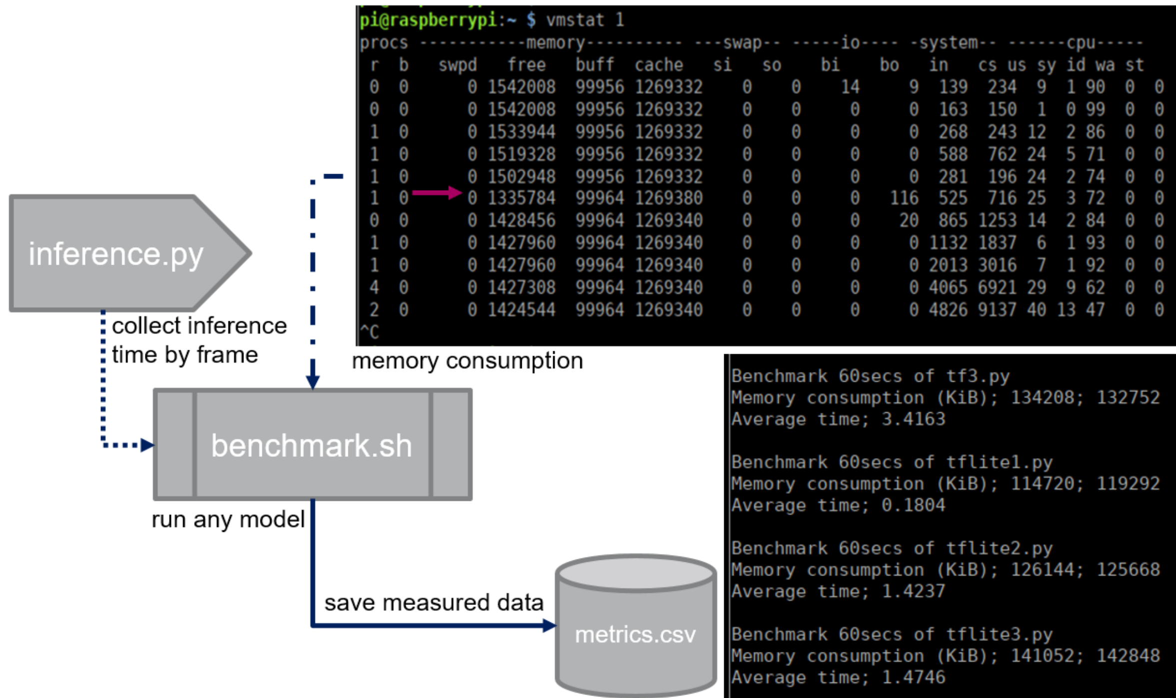


Figure 24: Autonomous benchmark script architecture

metrics can be performed in sample intervals, which allows the observation of the system activity in near real-time.

A sample interval of 2 seconds is given to VMstat tool, hence memory consumption is collected every 2 seconds and stored. When the inference algorithm finishes, an exit code is returned by it and caught up by the evaluation tool. With the inference algorithm finished, the script averages memory consumption during the execution time and subtracts to the initial values when the algorithm was not running, producing an average value of the consumed memory by the DL workload.

The evaluation of execution time of the inference phase cannot be retrieved exclusively by the shell-script evaluation tool. This is because the tool cannot access algorithm runtime stack of inference phase only in order to calculate the spent time on a specific part of the algorithm.

To bypass this limitation, inference time is calculated by the algorithm itself. And per iteration, for processed frame, the time spent on predicting the data is computed and printed to *stdout*.

The evaluation tool can now check the time being spent predicting the information on each frame, by accessing the *stdout*. Each retrieved inference time is stored and when the exit code happens, the average inference time and memory consumed by the workload is recorded on a logfile. This file can be used later for future evaluations of other algorithms or algorithms running on different frameworks, easing the generation of bar charts of the retrieved metrics.

3.3 ML INFERENCE ON RASPBERRY PI

An extensive analysis of the developed solutions this platform execution is detailed on this section, as well as the challenges faced while deploying the algorithms onto this system. It is also presented what libraries were used and why it were chosen. The multiple implementations developed, the deployment on multiple libraries and frameworks, and the challenges are also described in the process.

3.3.1 *Platforms Decisions*

Libraries

The main processing unit of this platform is a ARM Cortex-A CPU, and the library TensorFlow was chosen to provide a ground-truth benchmark since this library is the primordial and the most used on Machine Learning context. This library also has an extraordinary maintenance team always upgrading the packages and evaluating its performance.

TensorFlow Lite is another library that will be tested. It is a variation of TensorFlow, highly customized and developed to maximize the performance on embedded devices. To run faster, TF Lite quantizes the DL models and takes full advantage of the device resources, using the advanced Arm NEON optimization and operations of the ARM CPU quipped.

Evaluations were also performed on Arm NN, an inference framework developed by ARM. Arm NN was built to run on ARM CPUs only, and likewise the TensorFlow, this framework is highly customized to perform tasks efficiently on devices with ARM CPUs.

PyArmNN, a python interface to the Arm NN framework was also used and tested running these workloads.

3.3.2 *System Tuning for ML Inference*

Four different implementations of the inference algorithm were developed, each one running on each inference library or framework. The architecture of each implementation is described on the next sections.

TensorFlow implementation

The first implementation created to be deployed onto the the Raspberry Pi, was purely coded in Python using the TensorFlow library.

The main responsibility of the algorithm is to real-time predict every frame captured by the RC vehicle, to adjust the vehicle's steering angle while following a predefined trajectory.

This algorithm is composed by three phases, represented in Figure 25: the preprocessing phase, where all the setup and preprocessing takes place before sending the data to the deep learning model; the inference phase, where the predictive model uses the two output neurons to predict two distinct values — steering angle and a number representing the road lane visibility; and the communication phase, the component that interacts with the micro-controller of the vehicle, giving direct instructions to the vehicle — speed and wheels steering.

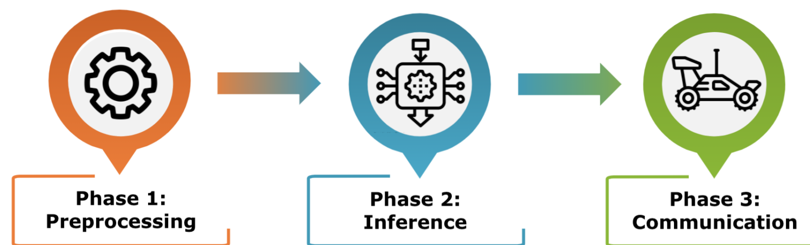


Figure 25: The three phases of the inference algorithm

Before the frame can be fed to the trained deep learning network, an important preprocessing must be executed. The first stage is responsible for loading all the necessary modules and to import the pre-trained model into the system's memory.

After the model had been internally loaded into Python memory space, the TensorFlow interprets the data as a graph of instructions, being ready to predict the data from an input image.

Then the camera, which is connected to the Raspberry Pi is activated and set to record image data on a 640x480 resolution. A thread is then launched, responsible to infinitely record frames of the video.

Later, each frame is sent to a web-browser, which the user can access and watch the live-feed of the vehicle's driving. Also each frame is resized to a resolution ten times smaller, so it can be passed to the inference phase, with reduced pixels, improving the performance of the algorithm by using less memory.

After that, inference phase takes place, real-time predicting each frame with a resolution of 64x48, that the previous thread is post-processing, as illustrated in Figure 26. This step is processed in real-time in a loop upon each frame being captured and post-processed by the thread assigned to. Every frame is sent to the CNN model, which processes its given input and through the pre-trained weights predicts the data.

Inference phase will generate two meaningful values as mentioned earlier — the steering angle and a value representing the percentage of the road lane visibility.

The third and last stage establishes a communication with the microcontroller, to send instructions to the RC vehicle. If the road lane is visible in each frame, the steering angle predicted is sent to Arduino, which communicates with the servo of the vehicle, spinning

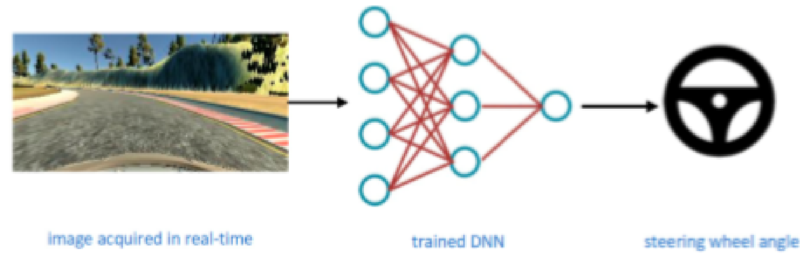


Figure 26: Inference phase: steering angle prediction

the wheels according the angle predicted on Raspberry Pi. If the road lane is not visible, the angle predicted is discarded. In this case, an order is sent for the vehicle steer the wheels aggressively to the left or to the right, upon the previous known direction that the vehicle was driving.

TensorFlow Lite implementation

The second implementation of the algorithm uses the TensorFlow Lite library, which is highly customized for embedded devices.

This alternative follows the same 3-step architecture defined on the TensorFlow implementation: preprocessing phase, inference phase and communication phase. The main changes against the previous implementation happens on the first phase.

This time, the preprocessing phase applies several optimizations to the graph of instructions imported to the Python environment. After the model had been loaded, some TensorFlow Lite routines are set and configured so the model can be optimized targeting the architecture of the Raspberry Pi hardware, specifically ARM configurations and pruning nodes, reducing some mathematical operations from the network.

Several optimizations are provided by TensorFlow Lite, but not always can be applied. In fact, multiple optimizations will bottleneck the algorithm and deteriorate the performance. Post-training quantization of the model could be set, but the Raspberry Pi ARM processor cannot handle native half precision operations. Triggering this operation will massively slow down the code execution because the system will unpack the single floating point values to 16-bit size, but then the CPU will still have to convert it again to 32 bits to compute the arithmetic operations and then convert again the output values to 16 bits (Figure 27).

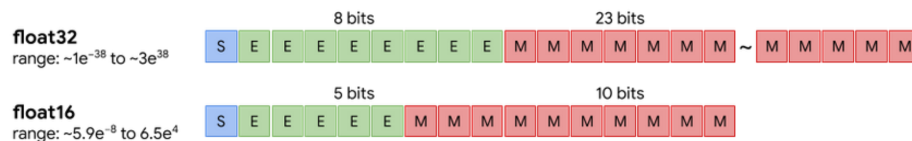


Figure 27: Post-training quantization

There are several post-training quantization options that can be applied. The Table 7 presents these multiple choices and the benefits provided by quantizing an already-trained float TensorFlow model.

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 7: TFLite post-training quantization options

After all optimizations, the succeeding behaviour of the preprocessing phase was identical to the TensorFlow implementation. A thread is spawn which manages to capture images frame by frame. The images are sent to the web-browser to live-feed the driving. Images resolution is decreased to be fed to the deep learning model.

The inference phase did not require much structural changes from the developed scheme of the previous library implementation. One optimization could be performed here, but the hardware could not pace the change. If the ARM CPU had support to native half-precision floating point operations, the frame could be scaled to FP16, scaling the image to half of its size. By halving the number of bits representing each pixel of the frame, would prepare the model as well perform half-precision inference.

Since the Raspberry Pi does not support enabling all these advanced optimizations provided by TensorFlow Lite, enabling the quantization techniques turn the algorithm slower rather than not using these techniques at all.

Last step of the three-phased inference algorithm does not contain any changes from previous application.

ArmNN implementation

This algorithm was developed using the open-source inference engine framework Arm NN. The selection of this library comes due to its support to ARM CPU's with the NEON vector extension, accelerating this type of ARM NEON devices. Multi-core support and the exploitation of ARM CPU advanced parallel features strongly advises to a highly scalable algorithm, which is a must for these ML workloads.

ARM provides its machine learning framework written in portable C++14. However the already developed modules on previous implementations, were written in Python, due to the libraries used. These modules allows to enable the Raspberry Pi camera, and helps to collect training data and launches the web interface to manually control the RC vehicle or to watch the first-person live feed of the autonomous driving mode.

Migrate all these modules to C++ would take some time and a solution was found out to keep both C++ Arm NN implementation and reuse the Python modules that communicates with the vehicle and manages all real-time data.

Initially the whole core module was planned to be migrated and re-written to C++. However, this process showed to be too time consuming, since this module had several components and would take longer if converted to the another language, rather than merging the services, specially with a fully web-browser developed in Flask. Using C++ to rewrite the web-browser would had been stepping backwards rather than forwards.

To overcome this drawback, the alternative found was to create an abstract compiled C++ library that only replaced one phase of the inference architecture instead of the whole modules.

The developed library would replace only the second phase of the architecture of the inference algorithm (Figure 25). This library was built to be external callable. In another words, was compiled into a Linux object so it can be executed from Python environment, since it can import Linux objects libraries. Its main function is to build a custom object that will keep all the Arm NN session needed to predict multiple images and to predict a frame, like so, this C++ object will be kept alive until the Python session dies. This strategy keeps from Python not to generate the engine over and over, every time it loops each frame.

Three methods are available to be externally launched: the *ArmnnEngine*, that creates and keeps the Arm NN engine session alive; the *startEngine*, that allocates all data and metadata to efficiently load the weight model, keeps the session alive all the way through every frame inference, reserves the image passed by and allocates the results of the image prediction; this method has one argument, a string that defines the name of the pre-trained weight file to be loaded; and *inference*, that predicts the image passed in on its arguments and saves the two output values on the addresses passed as arguments.

Arm NN provides several parsers to load neural networks defined in multiple formats, such as TensorFlow and TensorFlow Lite, among others. It was used a TensorFlow Lite FlatBuffers weight file as the model to be loaded. Like so, this implementation takes usage of TFLite parser armnnTfLiteParser, which is a library to easily import the weights file into the Arm NN SDK engine.

The function on the algorithm responsible to start the inference is prepared to predict the frame with some optimization sets targeting the platform from where it is being executed. In this case, the code will exploit the ARM CPU advanced operations and activate SIMD extensions with ARM NEON technology.

The shared class library functionality was explored to be accessible by the Python environment. Portability of the developed code to all programs, running Linux, was achieved by compiling it to the shared library. This allowed the library to be linked to Python environment at run time. In that way, Python can use the library without specifically contain it in its

memory address space. A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory.

The library is automatically linked into a program when the program starts, and exists as a standalone file.

A C++ Boost library was initially integrated to build the shared class that is automatically linked into a program. Boost provides a Python abstract interface — the Boost Python Module — to establish a communication between Python and the API. This first solution revealed some run time dump memories correlated to the custom object sharing among another programs memory.

A new alternative was found to circumvent the memory leak issue. To fully use the C++ library from Python standard library, the ctypes module was used. Ctypes is a foreign function library for Python that provides C compatible data types. Its local memory can be accessed, allowing functions to be called or variables to be taken.

The architecture of this developed C++ library that communicates with the Inference module developed in Python is illustrated in Figure 28.

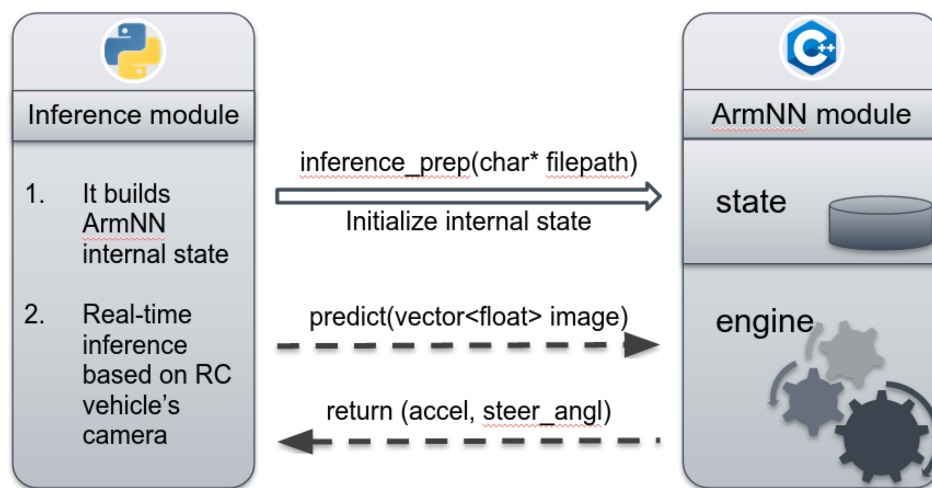


Figure 28: Communication process between Python and C++

With the library set, the *armEngine* object can be callable from the Python session. A Python class must be created to modulate all data from the shared library. This Python class has the reference to the three methods of the API and the possibility to send and receive the output data to Python declared pointers. To allocate pointers on Python environment, ctypes was used, since Python does not offer native support. It is possible now to invoke the methods defined on the API through this abstract class created on Python side.

API was built to simplify the Arm NN implementation and easily incorporate the Arm NN engine as well. Developing the Python client side of the inference algorithm turned to be effortless, since the API perfectly fits on the previously developed 3-phase stage inference algorithm.

The first stage of the (Python) algorithm, dictates the declaration of the Arm NN engine abstract class to interact with the shared library API. Follows the model file loading, using the defined method — *startEngine* — which creates and allocates all the session to Arm NN perform inference on the C++ shared library. The frame responsible to record the frames is then launched as it explained on previous implementations.

Second phase — the inference — can be perfectly adapted from previous inference phase implementations. The preprocessed frames captured by the thread are sent to the Arm NN engine, replacing the TensorFlow *predict()* method for the defined one on the API — the *predict()* method illustrated in Figure 28. The C++ *predict()* method is then called, which takes the frame as argument, linking with the shared library that computes the given image and outputs the two desired values. The predicted data on the ArmNN module is then sent to the Python Inference module (using the *return* method, also illustrated in Figure 28).

The third and last phase is defined on an analogue way as previous implementations. The two previous predicted values are now sent to Raspberry Pi. This process is cyclic repeated, predicting frame by frame taken by the capture thread.

PyArmNN implementation

This section contains an extra developed version of the algorithm using PyArmNN, which is a newly developed Python extension that provides an interface to Arm NN SDK C++ API. The source package is platform independent, but because it runs on top of the Arm NN engine, it requires full installation and compilation of the Arm NN.

Being the library written in Python, it gives a great possibility of re-implement the previous full Python developed alternatives. As most developed code is in Python, the advantage of using PyArmNN resides on a trivial development of the code. The implementation will be similar to the TensorFlow and TensorFlow Lite versions following perfectly the defined architecture.

Because this Python library is an interface/API to Arm NN engine, there is no need to develop and manage a connection to an abstract shared library like the previous Arm NN implementation, since the library manages that for us.

As the main topic of the this dissertation is inference performance, there is an urge need to test and evaluate the Python library against the pure Arm NN implementation. Thus it will be possible to analyse how it deals with memory management and check for possible internal optimizations on communication with C++ API, since it uses the same Arm NN backend.

Developing this implementation revealed not to be very costly since it only required some changes from the prior pure Arm NN implementation, following the same 3-phase architecture.

The preprocessing phase suffered some changes regarding to the custom object that was holding the whole Arm NN session. As now, exists an already developed Python library that

connects to Arm NN engine, it was created a similar object but using PyArmNN methods. Inside the class was created the same methods as the shared library — *startEngine* and *inference*.

startEngine method was reprogrammed to successfully hold the frame passed by in and to allocate all the tensors needed for the prediction process and to store and predicted data. Also it loads the weight file named on its argument by using the TFLite parser to import the TensorFlow Lite Flatbuffer model file.

The defined *inference* method, as the name suggests, takes the final interaction with engine by sending the frame to Arm NN SDK. The engine processes the image and the two predicted values are then returned. The data is saved on the object to be accessed later.

Like the prior implementation, the first phase starts by creating an Arm engine session, allocating the object previous described. Then the TFLite parser takes in to load the model file. The capture thread is now ready to be launched and start to capture frames from the Raspberry Pi's camera and post-process it.

On the inference phase, the loop is started, where each ten times smaller received frame, by the thread, is taken by *inference* method. Prediction takes effect and the steering angle of RC vehicle and a confirmation whether the road lane is present on that frame is predicted.

Third phase deals with the embedded system communication, sending explicit orders to the vehicle. Received orders are interpreted, adjusting the car steering angle to follow the road lane. The iteration of the loop finishes and loops back again starting the second phase that receives the next frame predicts the image, send the orders upon the predicted data, and the cycle goes on and on, infinitely driving through the road lane and predicting each frame captured.

3.3.3 *Challenges*

To follow a path

Deploying these deep learning algorithms on the Raspberry Pi platform did not reveal to be an easy task, which turned be more time-consuming than the algorithms development itself.

In this section will be described the deployment process of the multiple implementations developed and the major issues faced alongside this process.

TensorFlow and TensorFlow Lite were the unique libraries that did not gave any problems when deploying the developed algorithms. Both Google's libraries proved to be very efficient on the deployment stage, even on embedded systems. The extensive documentation and the great community helped to streamline this process, as well.

The Figure 29 shows the inference times of three different pre-trained networks — Linear Model, CNN_noaug and CNN_aug — executed on TensorFlow and TensorFlow Lite.

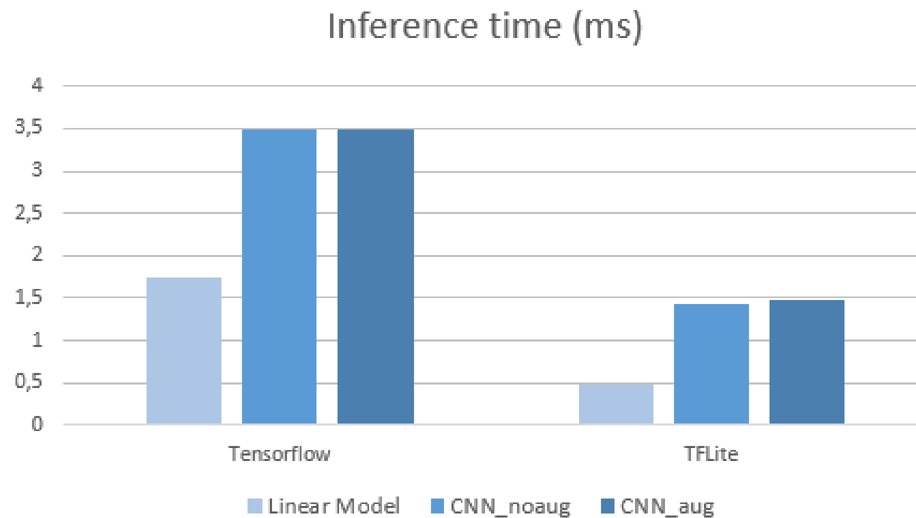


Figure 29: TensorFlow and TensorFlow Lite inference times with different file weights

TensorFlow Lite presents an higher efficiency running all the three models than TensorFlow, resulting in execution times more than 50% faster compared against TensorFlow.

The obtained results for each library showed itself to be very acceptable for the system being tested. A full detailed explanation is given in the next chapter.

Native Arm NN implementation is from far, the engine that gave more technical problems on the deployment. After the full development of the algorithm, tests presented some drawback regarding to the previous results. This library is the most optimized one among the libraries tested on Raspberry Pi. It supports a wide-range of hardware optimizations and exploit the ARM CPU at the most. Yet, the results obtained did not show any improvements, with spikes on inference times, as presented in Figure 30.

These results are far from being acceptable because execution times got a way quite higher than TensorFlow and TensorFlow Lite results. Arm NN could be a little slower compared against previous results, but these tests exceeded an upper roofline margin and cannot execute four times slower than the others libraries. An extensive debug and time was devoted to find the potential bottleneck of this implementation.

Multiple tests and theories were created to find the potential solution of this drawback. Implementation was checked and re-coded, the whole Arm NN modules and components were fully reinstalled and redeployed, the algorithm was carried with multiple load tests of multiple use-cases and inference executions were traced and profiled using appropriated performance tools.

After a code verification and multiple load tests performed, the algorithm kept behaving slowly for all given inputs, not showing any signs of improvement. To amplify the use-case tests of the algorithm, different configurations were tested.

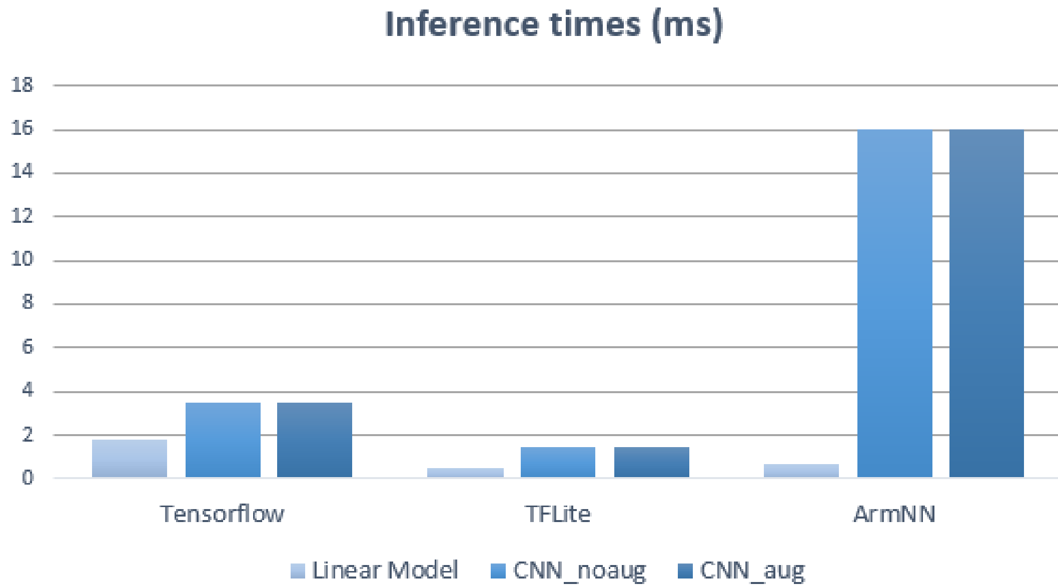


Figure 30: Arm NN inference times with different file weights

Initially, the algorithm could only predict frames with resolutions of 64x48 pixels. To extend its capabilities of predict multiple frames of different sizes, a change was made. The neural network architecture was not modified but the training data was enhanced. The original dataset that contained a set of images of the RC vehicle driving was duplicated, and it was processed a resize of four times larger to the duplicates images, generating a new set of identical images with a resolution of 256x192 pixels. This process is designed by data augmentation and for this use-case it aimed to support two different image resolutions: 64x48 and 256x192 pixels.

This process of trying inference predictions with four times larger inputs aimed to confirm or discard the theory that Arm NN could only enhance its engine on heavier workloads.

So the network was retrained, generating a new pre-trained model file that represents the neural network capacity to predict larger images.

When evaluating this new network on Arm NN, the results showed, again, no improvements at all. In fact, as illustrated in Figure 31, the algorithm slowed even more compared to against both TensorFlow evaluations.

This first try to find the bottleneck did not reveal to be efficient, since the model should have performed around the 16-21 ms per frame, like TensorFlow's executions. Despite the new model, *CNN_noaug_HQ*, be bigger, which could be more ideal to optimizations appliance, inference times got even worse. Compared to ideal execution times, this Arm NN implementation, executed seventeen times slower than the expected.

Consequently, these tests revealed that the problem does not resided on the model structure and on the presumable efficient data management by Arm NN on heavy ML workloads.

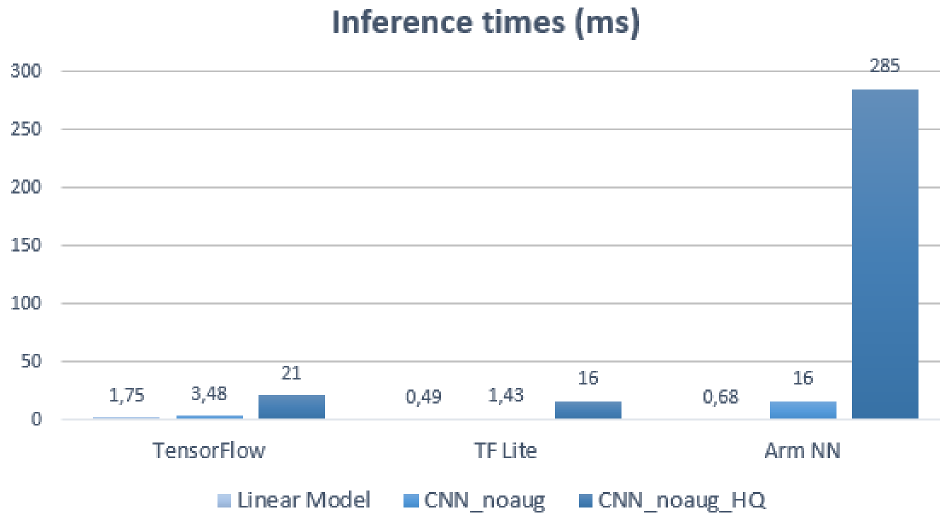


Figure 31: Arm NN inference times with different file weights

Profiling ML workloads

Given the difficulty to find the nature of the inference times taken by the Arm NN implementation and given the degraded performance, a profiling tool was used, the Flame Graph.

Flame Graph was chosen to aid on finding the potential bottleneck during the runtime of the algorithm. The tool can transform profiling logged data of programs into images, visually representing the profiling data.

The tool can generate Flame Graphs from profile data that contains stack traces, collected by different tools on multiple operating systems. The Table 8 presents all the operating systems and profiling tools supported, from which Flame Graph can convert the data into easy readable graphs.

Operating System	Profiling tool
Linux	perf, DTrace, eBPF, SystemTap and ktap
Solaris, illumos, FreeBSD	DTrace
Mac OS X	DTrace and Instruments
Windows	DTrace and Xperf.exe

Table 8: Operating systems and profiling tools supported by Flame Graph visualizer tool

With Flame Graph, several metrics of the system can be fetched easily and can be interpreted through the generated Flame Graphs. The x-axis of the generated graph shows the stack profile population, alphabetically sorted, and the y-axis shows stack depth, counting from zero at the bottom. Each rectangle represents a stack frame. The width of the frame represents the time spent executing a function or method. The top edge shows what is on-CPU, and

beneath there is its ancestor. The colors are not significant, since it were picked up randomly to differentiate frames. Gregg (2016)

Flame Graph supports a wide range of metrics, such as (i) memory usage; (ii) the off-CPU performance issues, namely the time spent by processes and threads when they are not running on-CPU; and (iii) the CPU usage, to check if a routine task is busy.

Figure 32, shows the Flame Graph of the Arm NN implementation, running the CNN model that predicts frames with a resolution of 64x48 pixels. This profiling was performed to build a simple baseline, to easily compare against future versions of this engine implementation.

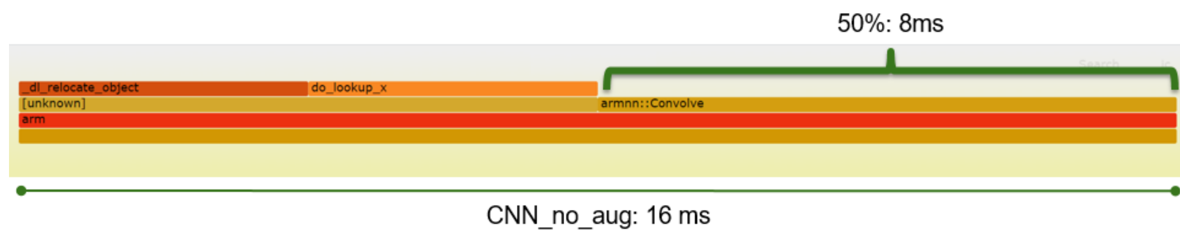


Figure 32: Arm NN inference times with different file weights

On the above Flame Graph (Figure 32), it is possible to notice that half of the execution time is actually spent doing math convolutional operations. Roughly one quarter of the time is spent on memory allocation and the remaining quarter is spent doing some internal functions inside shared libraries loaded by the Arm NN engine. This indicates that multiprocessing on the convolutional function is not happening. Google's libraries are taking roughly 1.5 to 4 ms to compute the whole algorithm, instead of Arm NN engine that is taking 8 ms only to do the convolutional work.

The curiosity of this aspect is that due to the ARM CPU architecture, this indicates the problem might be lack of multiprocessing and due to the non-exploitation of advanced ARM optimizations. This is because the algorithm is taking around 8 ms to predict a single frame and the average prediction of TensorFlow's versions are taking approximately 2 ms, that is, four times faster. This is also the number of CPU cores present on the Raspberry Pi 4, which indicates it is only using one single core to execute Arm NN engine, depriving the algorithm of multiprocessing capabilities.

An identical evaluation was performed on the implementations that predicts the four times bigger frames. Figure 33 presents the Flame Graph related to the profiling of a 256x192 image prediction. As expected, the same convolutional operation is also taking too long, spending 55% of the algorithm execution time on this method. The test also suggests the algorithm is not enabling multiprocessing capabilities, as the percentage of the time spent on the algorithm is related to ML math operations, like convolutional operations.

The performed tests suggest that the performance issue of the developed Arm NN inference version does not resides on a technical bug in the implementation, but rather in the inability of the engine to explore multiprocessing capabilities.

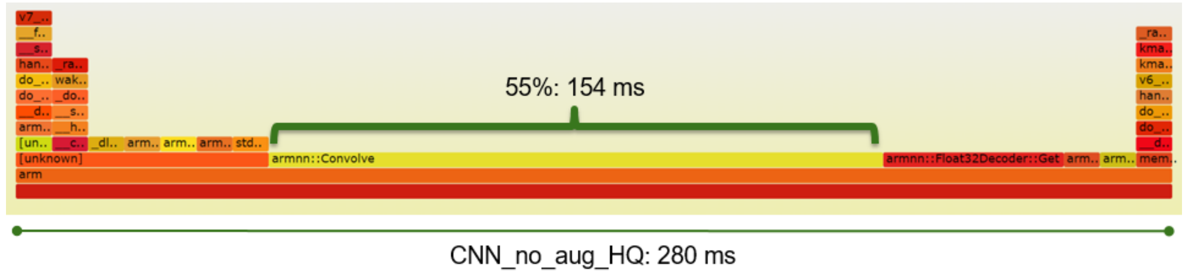


Figure 33: Arm NN inference times with different file weights

To cover and discard another use-case, a full fresh setup of the Arm NN engine was done again. Before this process, the Raspberry Pi operating system was re-flashed as well. A new version of Raspian OS was installed, following a 64-bits architecture version. This OS installation was executed to test the compatibility of the Arm engine running on 32-bits platforms, since initially the 32-bits Raspian OS version was deployed. This analysis covered the possibility of Arm NN engine only support 64-bits architecture, exploiting the full CPU capabilities on this version only.

The installation of this library on Raspberry Pi consisted on manually installing several modules to successfully load ML workloads on ARM CPU. As illustrated in Figure 34, for Arm NN run applications using TensorFlow models on Cortex-A CPUs, it was necessary to reinstall Arm NN SDK and Arm Compute Library (ACL). Some effort was taken to drive this process since it was necessary to manually install multiple dependencies, which in most cases had to be done by cross-compiling the libraries for the ARM platform.

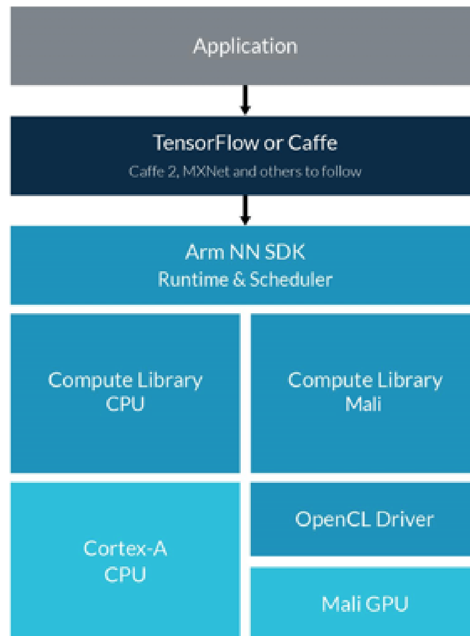


Figure 34: Data pipeline of Arm NN engine

A new training of the model was also executed on the 64-bits OS to discard any possible miss-configuration while training on 32-bits. The final results got were identical because the new generated model file (after the new training) was equivalent to the previous used one.

With the full environment set and ready to perform ML workloads, a benchmark was performed running Arm NN SDK on the 64-bits OS version. Represented in Figure 35, the implementation performed exactly with the same performance as on the 32-bits OS versions. This step settle down the theory the Arm NN engine could only speedup on 64-bits platforms and covered all installation phases to check important flags and configurations to correctly install the SDK on ARM architecture.

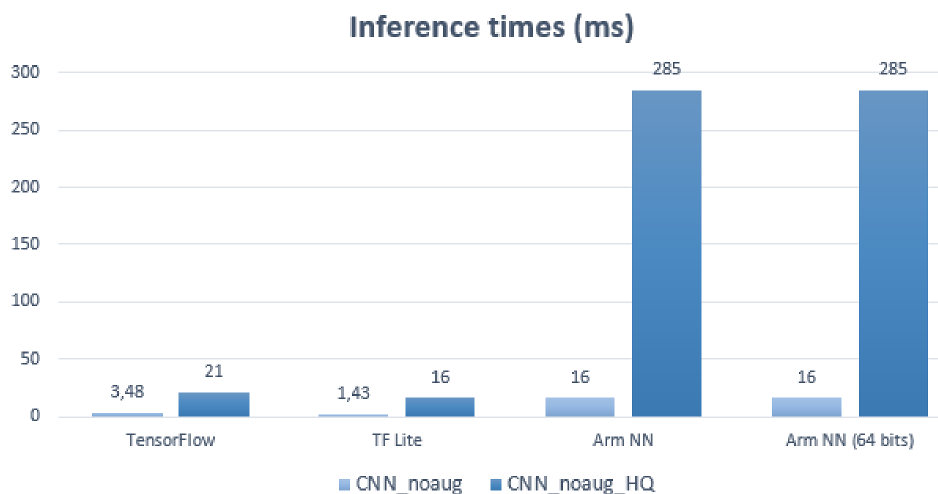


Figure 35: Inference times of multiple libraries including Arm NN 64 bits versions

Despite all these covered tests had not pointed out a possible solution to solve the bottleneck, a last profiling test was performed to analyse with more detail the model weight behaviour in the engine. A profile was performed in the inference phase of the algorithm execution, through the analysis of multiple model files generated by different neural networks.

The original CNN was modified into four multiple versions, generating four different neural networks. This nets were also trained with the dataset that contains driving logged data when the RC vehicle was manually driven.

The first neural network (*1-oconv*, Figure 36) was created by cutting off all layers of the original network, except the convolutional ones. This network architecture was created to evaluate the performance of the Arm NN engine while processing networks containing convolutional layers only.

Next, the original CNN network was readjusted and only the Dropout layers were kept in, aside from the final layers where the net diverges into two branches. Therefore, this network is very simple, as a consequence of the Dropouts, which discards a great number of neurons passing by on a single layer. The branches ares responsible for the output of the two values

through the two output neurons. This network, named *2-odrop*, is illustrated in Figure 37 and it mainly contains dropout layers.

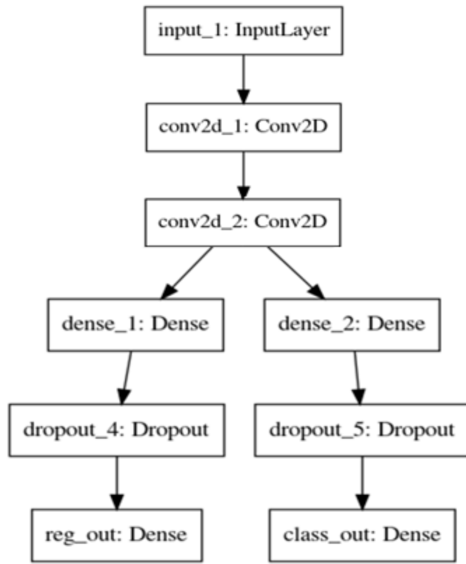


Figure 36: Modified neural net (*1-oconv*)

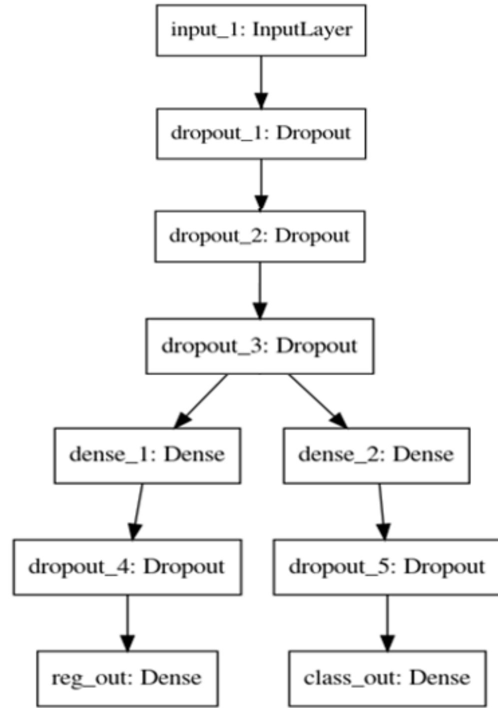


Figure 37: Modified neural net (*2-odrop*)

To evaluate the performance of the engine while computing a light convolutional neural networks, two more networks were created. Figure 38, presents a network, whose architecture was defined by a merge of the two previously mentioned neural networks. Thus for the creation of this net (*3-drop_conv1*), all original layers were cropped except the Dropout layers and the first convolutional layer. A kernel size of 1x1 was specified, defining a filter of width 1 and height 1 of the 2D convolution window.

Previous profiling tests depicted convolution layers are computing too slow. So, to perform an extended analysis of these ML workloads, convolutional layer parameters were reconfigured. The kernel size parameter was modified to profile the behaviour of this tuned operation. This leads to the fourth layer configuration as showed in Figure 39. The last neural architecture follows the same architecture of *3-drop_conv1* network, except the kernel size of the 2D convolution is 3 on this case.

With all these networks configured and set, each one was trained, producing four different trained models to be imported to the inference libraries being used.

Each model file exported by the training process was imported on the TensorFlow, TensorFlow Lite and Arm NN, to compare the multiple networks architecture against the several

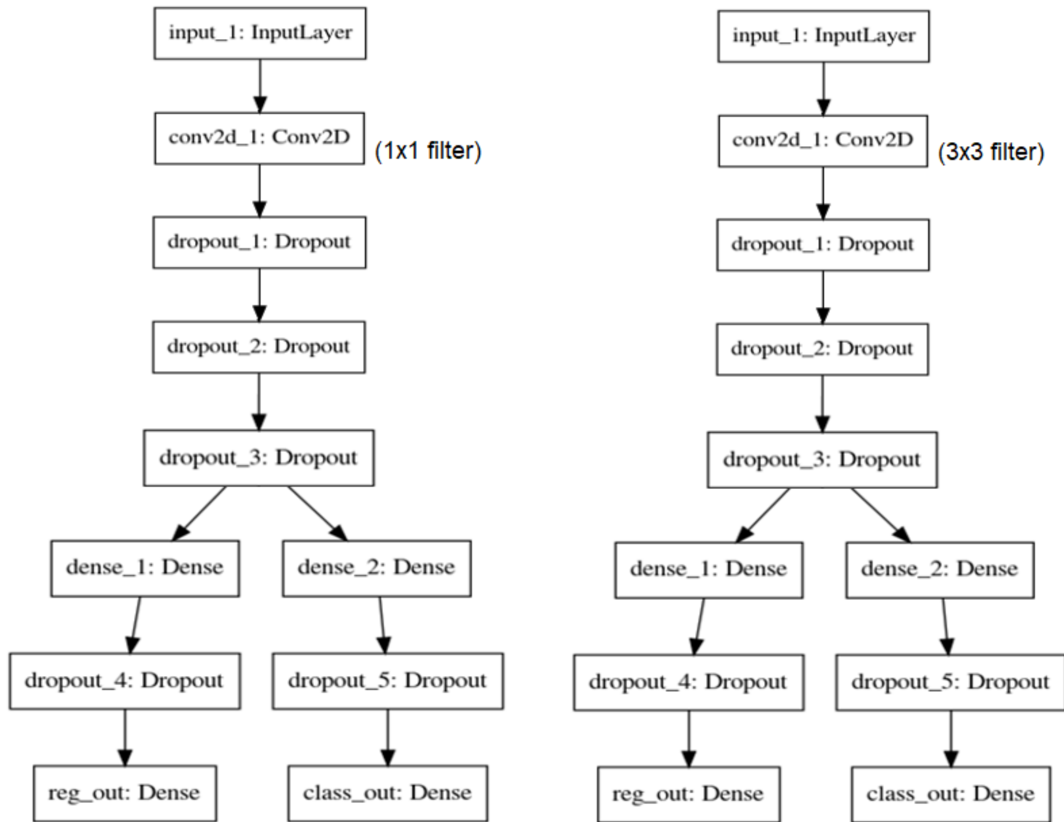


Figure 38: Modified neural net (*3-drop_conv1*) Figure 39: Modified neural net (*4-drop_conv3*)

inference libraries. The idea was to find the possible bottleneck on the implementation by checking what type of model fails to efficiently run on Arm NN engine.

The results of this test, illustrated in Figure 40, still pinpoints a lack of performance on the Arm NN engine. The average inference execution was about ten times faster on both TensorFlow's implementation rather than Arm NN.

Despite the creation of light networks (*2-odrop* and *3-odrop_conv1*), the performance on these was also poor. The model that performs only convolutional operations got even worst times, increasing its time comparing with the original CNN model.

As expected, the 2x2 kernel of the 2D convolutional window presents a slower inference execution (on Arm NN engine), when compared with the 1x1 kernel. The 1x1 convolutional neural network, which is mostly defined by dropout layers, also presents considerable worst times. However inference execution of this model on TensorFlow still presents pretty reasonable inference times.

To complement the previous benchmark, the two fastest networks were profiled to analyse what operations and functions are taking too much CPU time. The Flame Graph of Figure 41 points out that approximately 75% of the spent time, executing the Arm NN inference implementation, is doing absolutely nothing.

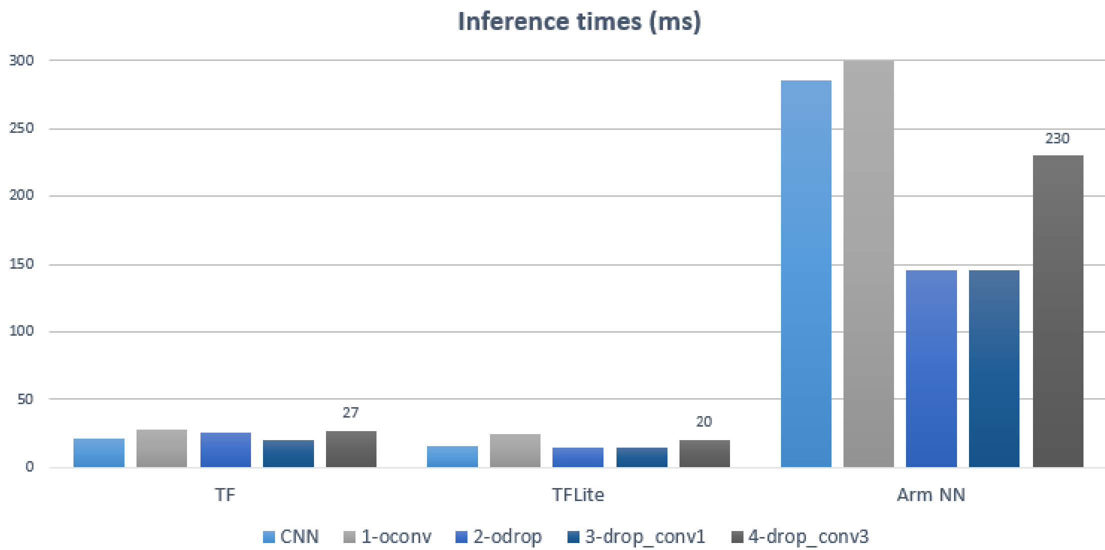


Figure 40: Inference times of tuned neural networks

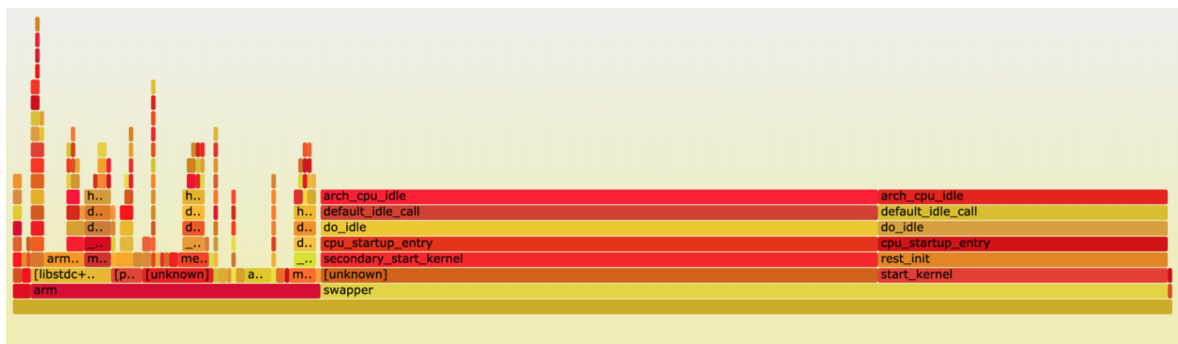


Figure 41: Profile of *2-odrop* neural network

After loading all the necessary modules and data necessary to perform the image prediction, the CPU gets stuck on a power save idle routine, *arch_cpu_idle*. Having the whole model processed in about 150 ms, leads to an unnecessary 110 ms spent on holding back the algorithm to output the predicted values.

The second Flame Graph (Figure 42) is related to the architecture that follows the layout of the previous network test with one additional convolutional layer at the start (*3-drop_conv1*). The bar chart indicates the algorithm is again taking too long on unnecessary loads and that is taking a considerable amount of time to compute the convolutional operation, as well.

Aside from the retrieved metrics by profiling the developed multiple inference implementations, the evaluation of these new neural networks architecture did not revealed itself to be very explanatory to find the bottleneck issue.

Consequently, only reprogramming the Arm NN implementation could solve these performance issues, since all planned tests had been covered. And as result, the only strong

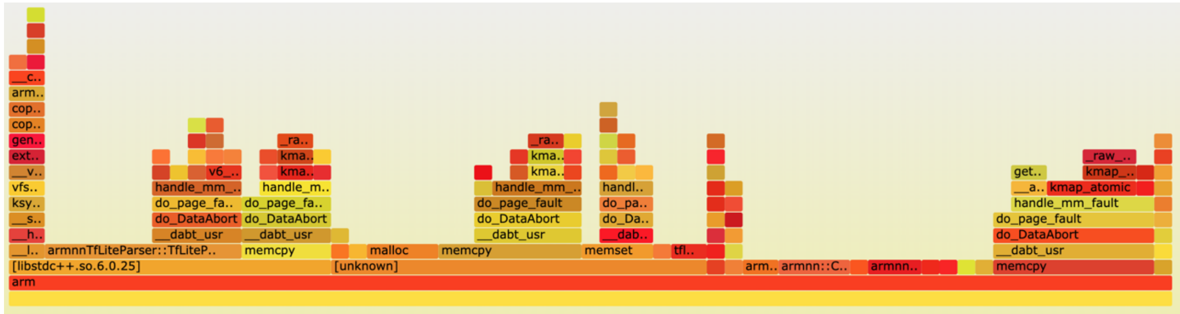


Figure 42: Profile of *3-drop_conv1* neural network

point and theory that could be taken was the algorithm was not taking any multiprocessing advantage.

Therefore, Arm NN documentation and its API was deeply checked out and tested, which allowed to discover that there were three important flags that could and must had been provided to the inference execution method.

There are three flags — *CpuRef*, *CpuAcc* and *GpuAcc* — which are intended to optimize developed Arm NN implementations. This options makes the compiler to turn on suitable hardware optimizations available, so the developed implementation can be highly optimized and exploit multiprocessing.

CpuRef stands for using a simple implementation using portable C++. This option is not suitable to be used in deployed applications because it will not use any ARM optimizations. This option is mostly for test purposes on non-ARM CPU architectures, like Intel CPUs. This flag will deliver poor performance results compared to the alternatives.

CpuAcc requires an ARM CPU on the system with NEON support, so the computing can be accelerated using the advanced ARM vectorization and parallel optimization techniques and enabling multiprocessing. It will only work on Cortex-A processors and a subset of Cortex-R processors. This is the flag that must be used on this use-case, since the deployment is being made on a Raspberry Pi 4 with one 4-core Cortex-A CPU chip.

Lastly there is the *GpuAcc* flag that requires a GPU on the system that supports OpenCL. This engine backend will only work on Arm Mali GPUs, so this flag should be activated to increase performance computing by using the GPU-cores of the system.

Unfortunately the developed implementation was not using the desired *CpuAcc* flag, hence all previous test results of the Arm NN engine showed off to be quite inefficient.

The Arm NN implementation was improved by using the correct optimization flag, hence the algorithm could be optimized and processed by the four ARM CPU-cores, using NEON advanced extension. A new evaluation was performed, and as illustrated in Figure 43, the bottleneck had been solved. The Arm NN engine backend can now efficiently predict each image, with a speedup of ten comparing to the previous results (Figure 30).

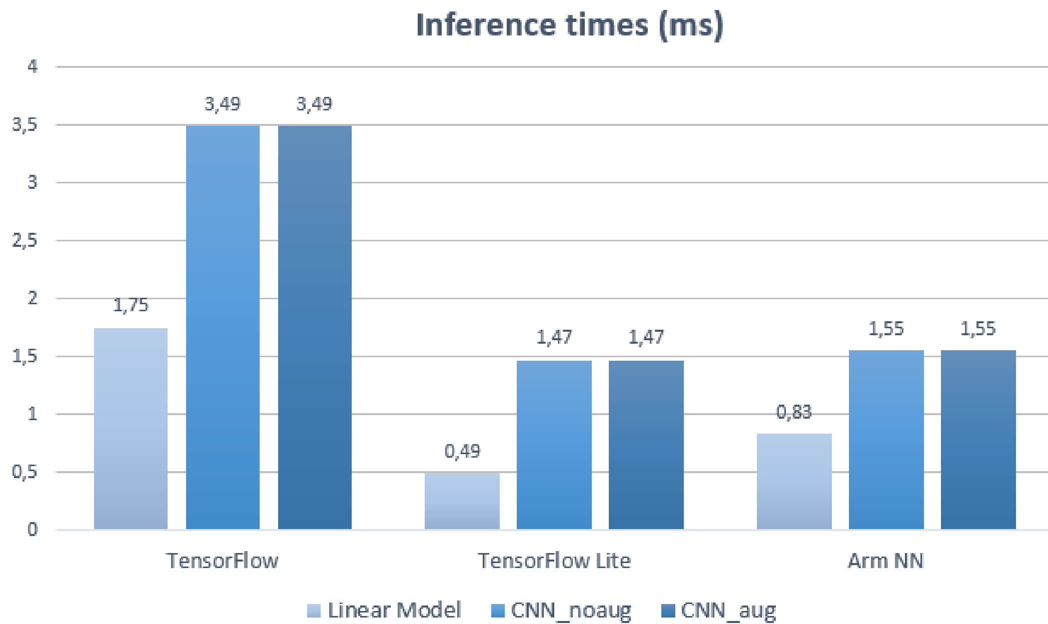


Figure 43: Performance of the developed implementations

To detect traffic signs

The FgSegNet model was fully benchmarked and evaluated to run on conventional systems, such as desktops. However, as on previous CNN models, the new DL workload has to be deployed on specific embedded systems, a Raspberry Pi 4 and a Jetson Nano.

To fully explore full capabilities of both embedded systems and to improve its efficiency while running these kind of workloads, the model was tested and evaluated also on different ML frameworks.

The new algorithm (Figure 19) was deployed onto the Raspberry Pi using TensorFlow, TensorFlow Lite, Arm NN and PyArmNN.

Deploying personal ML algorithms on embedded systems running on different frameworks or libraries, is not an immediate task. This process normally comes with multiple issues that need to be solved to properly adjust the model so it can be successfully deployed into these devices. This issues comes with versions incompatibilities, nonsupport of layers and even by hardware constraints.

Importing the DL trained model into Arm NN engine worked properly, so the deployment was immediate on the ARM engine. The deployment process for this inference engine is analogue to the CNN presented before, reason why this deployment will not be presented again.

However TensorFlow and TensorFlow Lite implementations were not immediate to deploy on the target system — Raspberry Pi 4 — which required additional steps to fix intermediary occurred issues.

TensorFlow

Memory issues

A Raspberry Pi 4 with 4 GiB of RAM was originally chosen and provided by Bosch for this project. However, external factors lead to the incapacity to use this model when deploying the traffic signs algorithm.

Although another Raspberry Pi 4 was possible to be used to perform this process, a 2 GiB of RAM model version was provided by Bosch Car Multimédia, S.A.. This new embedded device provides all the specifications detailed on Table 3, with the exception of the RAM, since it only have a 2 GiB RAM of capacity.

Nevertheless, using a restricted memory device, compared to previous model, played a major role to evaluate DL algorithms execution performance on constraint environments.

When using this Raspberry Pi 4 version to execute FgSegNet, the system crashed at the inference phase, when trying to predict an image. Giving the memory limitation of the present system, the algorithm was profiled and memory consumption was checked on multiple stages of the algorithm.

Just before the algorithm had started executing, the embedded device only has about 1 GiB of RAM memory available from a total of 1.8 GiB initially provided. This suggests that the OS is already consuming about 0.8 GiB of RAM.

During the algorithm execution, the memory usage starts to increase, decreasing the amount of available free memory. In every instance of the algorithm execution, the memory decreases down to about 80 MiB (Figure 44). This lower limit is reached when the algorithm tries to execute the inference phase, causing the algorithm to abort.

To complement the previous analysis, the same algorithm was also deployed on a personal computer with 8 GiB of RAM. On this new environment, the algorithm manage to run successfully, which allowed to observe that it consumed about 1 GiB of memory during its execution. That is also the same amount of free memory on Raspberry Pi, which strongly indicates it is not enough to properly execute the DL workload.

On the personal computer, a module that manages system resources usage was used to truncate the maximum memory the program can use. As on Raspberry Pi, a maximum limit of 1 GiB was set on the desktop. With these virtual constraints, the algorithm also crashed due to (virtual) memory limitations.

This was enough to conclude that the FgSegNet model provided, needs to use approximately 1 GiB of RAM to execute on TensorFlow.

```
       total     used     free   shared  buff/cache   available
Mem:   1.8Gi    1.2Gi    205Mi     57Mi       481Mi       548Mi
Swap:     0B       0B       0B
       total     used     free   shared  buff/cache   available
Mem:   1.8Gi    1.2Gi    189Mi     57Mi       487Mi       538Mi
Swap:     0B       0B       0B
       total     used     free   shared  buff/cache   available
Mem:   1.8Gi    1.2Gi    178Mi     57Mi       490Mi       530Mi
Swap:     0B       0B       0B
       total     used     free   shared  buff/cache   available
Mem:   1.8Gi    1.4Gi     83Mi     57Mi       367Mi       312Mi
Swap:     0B       0B       0B
       total     used     free   shared  buff/cache   available
Mem:   1.8Gi    1.3Gi    221Mi     57Mi       367Mi       451Mi
Swap:     0B       0B       0B
```

Figure 44: Samples of 1 second of system memory consumption. Purple box: the moment when the system rans out of memory

Layer issues

Later, a new Raspberry Pi 4 model of 4 GiB of RAM was granted again, which allowed to keep the deployment on this embedded system.

The algorithm was deployed onto the new 4 GiB device and managed to successfully run. However, the generated images did not produce the expected results. Each predicted image was producing a segmentation on the whole picture (Figure 45), instead of segmenting only the clusters defined by the traffic signs.

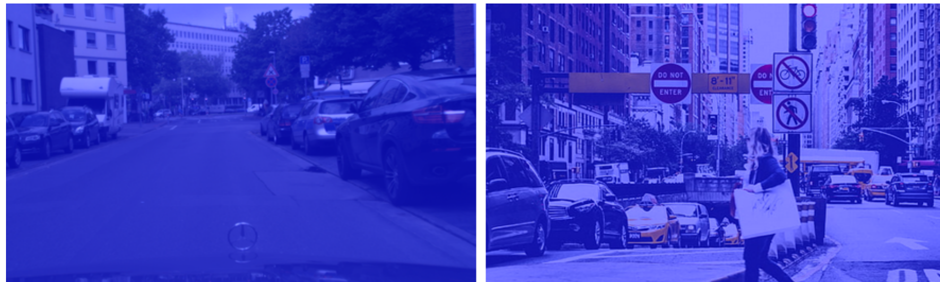


Figure 45: Bug on output of traffic signs inference

Given this wrong predicted data, the FgSegNet model architecture was extensively analysed in order to find a potential solution to fix this wrong predicted data.

When analysing the network, it was discovered that not all layers present in the model architecture are compatible with TensorFlow, since this library cannot parse Instance Normalization layers, which was included into the FgSegNet network.

To use this Instance Normalization layer, FgSegNet model must externally import it using another library called TensorFlow-Addons, which defines all intrinsic operations performed by this layer.

However, the additional layers existing into TensorFlow-Addons library can only be imported in TensorFlow version 2 or higher. But, at the moment of these tests, the most recent version of TensorFlow available and supported to embedded systems was 1.15. This version limitation motivates an improper loading of the Instance Normalization layer into the TensorFlow 1.15, causing major errors on the final predicted data.

To bypass this constraint, another layer had to be used to replace the one used on the original model because Instance Normalization layer used on FgSegNet model required at least TensorFlow version 2.

Two additional layers were found that could easily replace the unsupported one: (i) the *instance_norm*, a layer directly provided by TensorFlow 1.15 that could simulate part of the Instance Normalization behaviour; and (ii) the Batch Normalization, that can eventually perform identical to the Instance Normalization, leading to similar outcome results. These two layers were tested and evaluated, but this process required a redesign of the FgSegNet model, changing its original layer structure.

The redesign of the process generated two different FgSegNet models that diverge from the original one. Each model was generated by replacing the Instance Normalization with the *instance_norm* and Batch Normalization layers. This required that both models had to be retrained with Cityscapes dataset, repeating the training process.

Results of the tuned models revealed it can already segment parts of the image instead of the whole image. But predicted data of both models shows very poor results on the accuracy of the detected objects, as shown in Figures 46 and 47.

Figure 46: Inference on FgSegNet with *instance_norm*

The previous models could be successfully deployed on TensorFlow, yet the evaluation results clearly indicates these models must not be used in the context of this use case since they cannot properly detect traffic signs.

This problem was later fixed, given the constant upgrade of machine learning libraries and frameworks, nowadays. In the meantime, TensorFlow released support of its latest version 2 to embedded systems, which allowed to successfully setup the Raspberry Pi device with the required version, to successfully deploy the original FgSegNet, which carries the

Figure 47: Inference on FgSegNet with Batch Normalization

Instance Normalization layer. Figure 48 displays a proper traffic sign detection of the original FgSegNet model using the Instance Normalization layer.

Figure 48: Inference on the original FgSegNet (with Instance Normalization)

TensorFlow Lite

Initially the deployment of the model that aims to detect traffic signs was suffering the same issues as described earlier on TensorFlow. However, when using this library with Raspberry Pi 4, the 2 GiB RAM version, the memory consumption did not affect the algorithm execution.

On this case, the 2 GiB RAM capacity was enough to run the algorithm, since TensorFlow Lite optimizes the model being executed. One of the major optimizations is the compression of the model, which reduces the imported model by pruning the weights that do not play a significant role to predict the desire data.

Smaller models use less RAM when executing, which frees up memory for other parts of the application being used, which can be translated to a better performance and stability TensorFlow (2020).

Although the original model could be deployed on a memory constraint device, the initial inference results become quite similar as the bugged ones presented in Figure 45.

These inference issues were also being produced by the unsupported layer — Instance Normalization — on TensorFlow Lite versions lower than 2.

As performed with TensorFlow, the latest version 2 of TensorFlow Lite was officially released to embedded systems platforms, which fixed the inference issue by properly importing the required layer. The outcome of the algorithm is quite similar to the TensorFlow version, presented in Figure 48.

Arm NN

Deploying this DL workload on the Arm NN was trivial since the required pipeline to properly run inference models on this engine was already been covered earlier when deploying the CNN model.

With the latest TensorFlow Lite installed, which Arm NN uses to parse DL models, FgSegNet was successfully imported into this particular inference engine, which allowed proper predictions as showed in Figure 48.

3.4 ML INFERENCE ON HETEROGENEOUS JETSON NANO

On this section, a full explanation of the developed algorithms to run on Jetson Nano platform are be presented. Full details of the deployment on this embedded system are described and also how GPU-cores were exploited to run this particular program: an algorithm capable to predict two output values by processing frames in real-time.

3.4.1 *Platform Decisions*

Libraries

TensorRT is the inference library chosen to compute this specific type of ML workload. It is built on CUDA, NVidia's parallel programming model, and enables inference optimization for all deep learning frameworks. By running this inference engine, the GPU-cores can be enabled, which leads to an improved computing performance, when compared to an execution on CPU-cores only.

During the build phase, the NVidia engine identifies opportunities to optimize the network, and in the deployment phase TensorRT runs the optimized network in a way that minimizes latency and maximizes throughput Gray et al. (2017).

The Jetson Nano ARM CPU, a Cortex-A57, is an earlier version of the Cortex-A72 present in the Raspberry Pi 4. The ARM Cortex-A57 has a lower clock frequency and the size of the L2 cache is smaller. Thus, running any distribution of the TensorFlow library in the Jetson Nano ARM CPU would lead to a performance drawback. Jetson Nano ML workloads will

run slower on CPU, so it is fundamental to achieve an approach that uses GPU-cores to aid and support the heavy computations.

The specific NVidia inference framework was selected because it supports high-performance deep learning inference on this particular embedded system, providing a wide-range of GPU optimizations and accelerations by supporting fully-native FP16 operations.

TensorRT offers a C++ API and a Python API, which are both identical in supporting the same needs.

The main benefit of using the Python API is the ease of programming algorithms that perform data preprocessing and postprocessing because it offers a variety of libraries like NumPy and SciPy, which are very complete libraries to manipulate and calculate complex data structures.

When the use-case requires an higher level of safety within the application, C++ API should be used. For this RC vehicle use-case, a TensorRT algorithm was developed resorting to the Python API to easily reuse key components already developed in Python with the Raspberry Pi. This results on code reuse and on a reduction of spent time implementing this heterogeneous approach.

3.4.2 *System Tuning for ML Inference*

The implementation, running on TensorRT engine, follows the same architectural principles as previous CPU implementations (Figure 25). Hence, the developed algorithm, which runs on the Jetson Nano embedded system, also comprises the same defined phases: preprocessing, inference and communication phases.

There was no need to apply major changes to the communication phase since its backend does not particular depends on this inference engine.

A CPU processing approach is kept on preprocessing and communication phases to relief the GPU-cores of additional computational workload.

While CPU-cores are assigned to handle the computation on these two mentioned phases, GPU-cores are responsible to process, in parallel, the inference phase. Because a heterogeneous computing approach is present in this embedded system, inference phase must be fully reprogrammed to exploit full capabilities of Jetson Nano GPU.

As stated on earlier implementations, the preprocessing phase imports the model weights, allocates the inference engine and captures real-time frames from the attached camera. Its final role is to send each captured image to the inference engine (inference phase).

This particular NVidia inference engine only supports specific weight file formats to be loaded into TensorRT. So the first step to perform before enabling inference on TensorRT is to convert the pre-trained network into a TensorRT network. The easiest way to achieve this is to convert the model into one of the following formats: *UFF*, *Caffe*, *ONNX*.

The original model file, which was generated by training the network, was exported to a TensorFlow (*.pb*) format, so it was chosen the UFF parser.

The above parser offers an easy way to create the desired file from the *.pb* file. After the conversion into a TensorRT network, a weights file with format *UFF* will be generated, which is recognizable by TensorRT engine.

Preprocessing phase starts by loading all TensorRT and NVidia modules that will handle all the ML workload. With the supported weight file generated, the model can now be imported into the TensorRT engine.

This particular NVidia engine must be allocated firstly. One of its functions is to run the fastest implementation available of the kernel. Thus it is necessary to use the same GPU to build the engine and to run the inference, since the engine is built with specific tuned CUDA kernels, which targets the available GPU.

TensorRT engine has many other advanced properties that can be set, such as, autotuning parameters, batch size, defining the maximum memory allowed to be used by the GPU, and even the floating-point precision that the network should run. This last configuration allows to exploit the native half-precision support on this particular GPU, granting an FP16 inference mode. Running on reduced precision inference will lead to a significantly reduction in application latency, which is a critical requirement for most real-time and embedded applications.

Before the UFF interpreter begins to parse the imported model, the inputs and outputs of the model must be configured, allowing the creation of the TensorRT network. The UFF parser plays such an important role because it performs multiple important transformations and optimizations on the neural network graph.

These optimizations consist on eliminating layers with unused outputs, avoiding unnecessary computation. Transformations may also be applied where possible, by fusing certain layers (Figure 49), such as convolution, bias and ReLU, to form a single one (Figure 50). It is important to underline these graph optimizations do not modify the output results of the network, instead, they look to restructure the graph to perform the operations much faster and more efficiently NVidia (2018).

Back to the preprocessing phase, after the model had been imported, precision mode of the network can be chosen between single precision (FP32) and half-precision (FP16). Then TensorRT engine applies all possible optimizations mentioned earlier, generating an highly efficient internal network ready to perform inference.

With the engine ready, preprocessing phase finishes by allocating the host and device buffers for storing the input and the outputs of the network, respectively.

Three buffers must be created on both processing units — CPU (host) and GPU (device) — which each triple of buffers will be responsible to hold enough memory to store the input image and the two output values. A stream will also be created to establish a connection

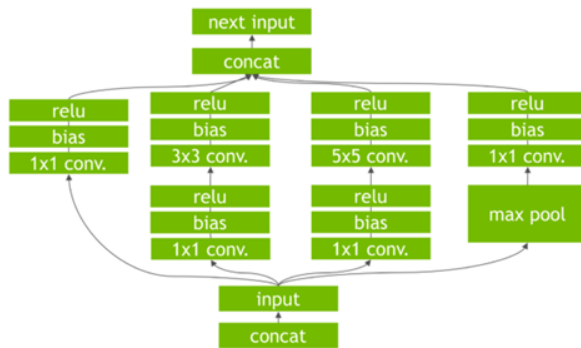


Figure 49: A convolutional neural network without any optimizations

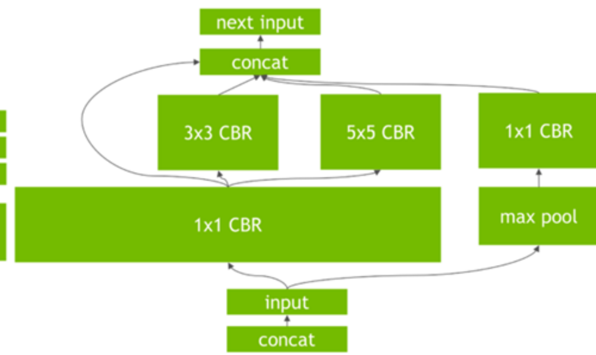


Figure 50: The convolutional neural network after several optimizations applied by TensorRT engine

between the host and the device, which will allow the information to be accessible for both processing units.

Particular TensorRT approach

The buffers, introduced earlier, must be allocated on each device to allow data transfers between the CPU memory and the GPU memory, which will process the data. Buffers must be allocated on CPU, where the data was firstly stored, and on the GPU to hold the data to be processed. Many platforms require this process because their acceleration devices do not share the address space with the CPU.

The Jetson Nano hybrid processor chip contains integrated GPU cores, which share the memory address space with the CPU cores. Figures 51 and 52 compare the architecture of a system with an external graphics card as accelerator with an architecture containing an hybrid CPU/GPU device, sharing the system memory space. In the first approach memory accesses may be faster, since the CPU-cores are not competing with the GPU-cores, but the time to transfer data between both memories may degrade the overall performance.

Nevertheless, a buffer on both host and target devices, and a stream supporting a connection between the two processing units must be explicit programmed, despite the Jetson Nano being used. This happens to turn TensorRT API independent upon the type of platform and its architecture being executed.

The TensorRT engine created earlier in the preprocessing phase, took information about the GPU being used, so the optimizations applied on the network were applied taking into account the type of integrated GPU architecture present on the NVidia Jetson Nano. In fact, the method that creates the triple-buffers, does not actually allocate the same memory on the device (gpu). Depending on the type of GPU architecture present, TensorRT creates references of the input and output buffers simulating a memory allocation.

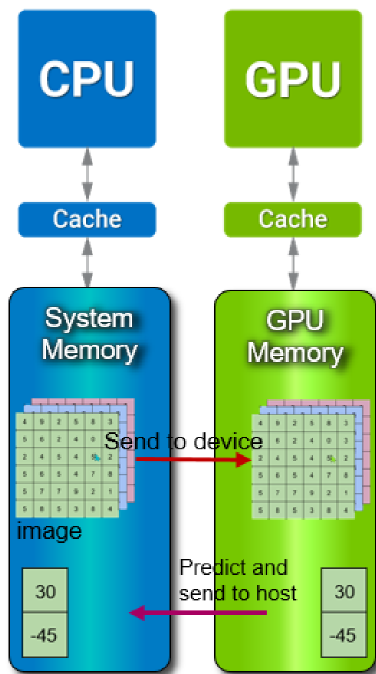


Figure 51: Discrete GPU architecture

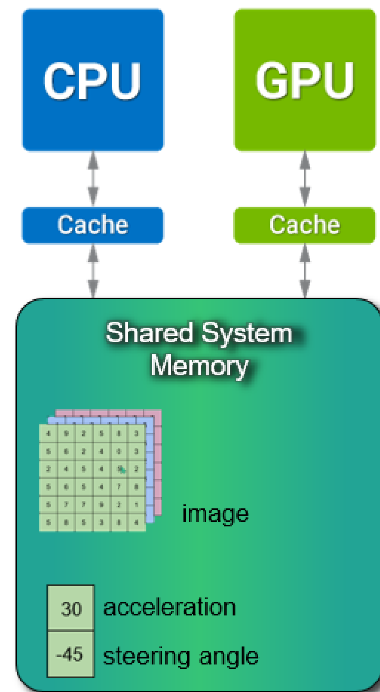


Figure 52: Integrated GPU architecture

Moreover, the stream created, also acts like a fake connection, mapping the values between the created references. Originally, the stream would be bidirectional to send back the predicted data to the CPU, as illustrated in Figure 51. In this case, the reverse channel of the stream, inserts the predicted data by the GPU into the original allocated buffer, on the shared memory.

With all TensorRT optimizations set, the algorithm is ready to effectively start. The thread (that was deeply explained on Raspberry Pi implementations) enables the Jetson Nano camera is activated, capturing the frames from the video that is being recorded, and sends the data to the next phase (inference phase).

Follows the inference phase, where TensorRT activates the GPU-cores of Jetson to predict the data. In real-time, the image captured and postprocessed by the thread is now accessed and stored on the network input buffer. The image is mapped to a reference created on the shared memory system, so the GPU can send it to its cache and start to predict the data from that image. Then each output value predicted by TensorRT is mapped again to the original network output buffers.

Any access to the buffers after the copy from the host to device (Figure 51), must be treated with extremely caution. This memory sending is asynchronous, which means that multiple data can be sent back from the device in parallel. A waiting routine must be called after the last function that sends the data from device to host.

Since Jetson Nano has a shared memory system (Figure 52), the data is not actually sent from GPU to system memory due to the shared memory system container, as explained earlier.

Thus any flaw occurrence originated by the nonexistence of this synchronizing routine is lower but not null, so this routine must be kept after the output buffer mapping.

The last phase — communication — of this cyclic operation takes over, sending the two predicted values from Jetson to the Arduino, which controls the vehicle (a fully explanation of this phase was also described on previous Raspberry Pi implementations).

3.4.3 *Challenges*

To follow a path

The deployment and configuration of the developed heterogeneous implementation for Jetson Nano was, by far, the most time-consuming one, which required an extremely caution analysis of each component and each task that defines the whole workflow.

Deploying and running this DL algorithm revealed itself to be a slower operation than expected. At first, the results seemed to be very reasonable, showing a speedup on inference time, comparing to Raspberry Pi inference times.

As illustrated in Figure 53, TensorRT performed even faster than the quickest Raspberry Pi implementation, executing in less than half of the Arm NN inference time.

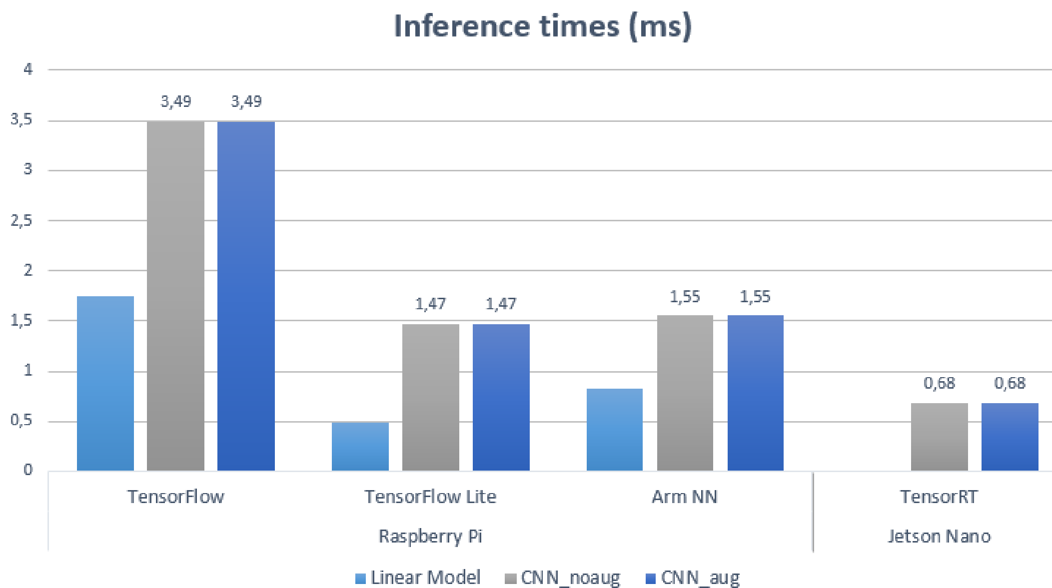


Figure 53: Performance of Jetson Nano TensorRT against Raspberry Pi implementations

These results seemed to be quite accurate for the NVidia platform, but a quick verification on the predicted values, showed the generated values were being wrongly predicted. Since the values were not being correctly predicted, a re-evaluation of the network performance was done.

For this evaluation, it was created a black image, for test-purposes only. This image used on multiple implementations and it was fed to the same convolutional neural network. Despite the test had been done on different libraries, the output results should be the same or pretty close[1], since the same (black) image is being fed to each network input.

Raspberry Pi multiple implementations produced the same output values after an inference evaluation, lets choose them as a baseline to the correct predict values (Figure 54).

However the output values on Jetson Nano, using TensorRT, was not the expected ones, producing values too far from the expected ones. The miscalculated outputs using the heterogeneous implementation are illustrated in Figure 55. It is possible to notice the TensorRT predicted a steering angle too different from the Raspberry Pi implementations, and eve the lane view value suffered some deviations.

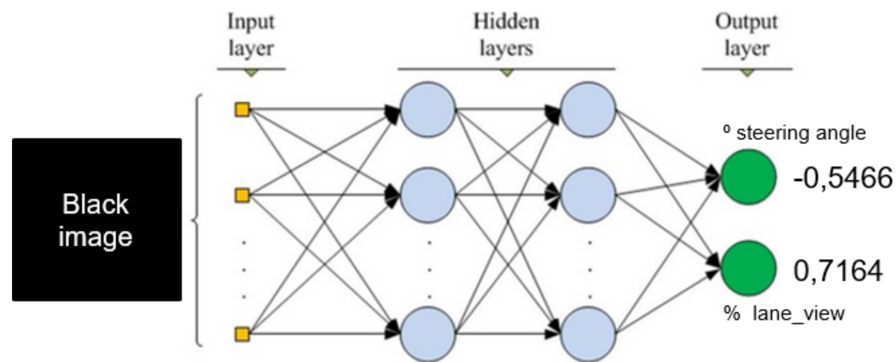


Figure 54: Correct output values predicted by TensorFlow, TensorFlow Lite and Arm NN

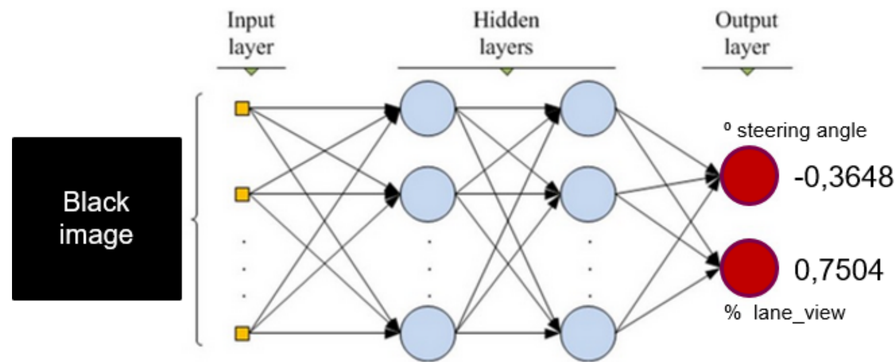


Figure 55: TensorRT wrong predicted values

Multiples paths were covered to find the solution for this inference issue. First, the conversion pipeline that parses the original trained model, and generates the final specific file format supported by TensorRT engine, was revised and debugged. As the issue could reside

1 A different outcome can occur if the weights are changed or optimized, such as changing the precision of the network (single-precision to half-precision, or vice-versa). Yet, although such optimizations introduces differences on the output values, they should not be too away and must be close to each other inside a narrow margin.

on a conversion process in between all the conversion pipeline that is necessary to import the model to TensorRT. Both tasks were revised.

Loading the weights file into the NVidia engine required two file conversions. The process starts by converting the original model file, which was exported to a TensorFlow/Keras format *(.h5)* into an optimized frozen TensorFlow graph. The exported frozen graph defined on a *.pb* file already contains some optimizations within. Then this *.pb* model file needs to be converted to such file extensions readable by TensorRT, as mentioned on the earlier subsection. This two-stage conversion pipeline is illustrated in Figure 56.

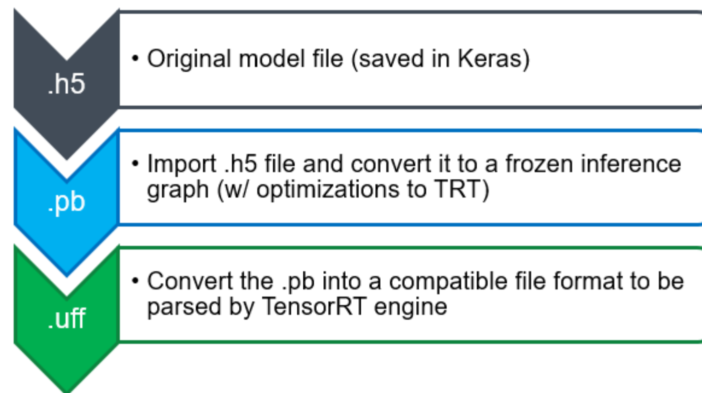


Figure 56: TensorRT conversion pipeline

The first conversion (*.h5* to *.pb*) was deeply analysed, and the *(.pb)* weights file was tested on different libraries. The results taken showed the same predicted output values, running on Keras and on TensorFlow, the same as Figure 54, as well. This concluded the first conversion stage is not generating flaws on the network architecture.

Lastly, the algorithm that generates the *.uff* file from the *.pb* model file was also examined. A converter provided by NVidia was used to discard any flaw on this implementation. The converter is able to parse a TensorFlow model and internally converts it to the required format by TensorRT. The results obtained were exactly the same as the previous mentioned in Figure 55.

Debugging the pipeline did not led to find any solution to fix the TensorRT issue. But this process assured the problem came after the *.uff* model file that was being imported into the NVidia engine.

This problem could occur during the final phase of internal TensorRT optimizations of the network. Issues coming from the non-support of some layers were discarded, since all layers that define the CNN are supported by TensorRT. This CNN is composed by convolutional, max pooling, dropout, dense and flatten layers, which all are supported by TensorRT.

To discard any potential problem on GPU memory during an inference execution, a profile was taken measuring different amounts of maximum memory being able to be used by GPU. Several inference profiles were executed on each one.

Nevertheless, the CNN architecture was tweaked and multiple versions were created to analyse the behaviour of each layer execution.

The strategies applied on debugging the Arm NN implementations to find the bottleneck issue were also applied on this solution. Consequently, five different neural networks architecture were created and trained aiming to find a version that successfully predicts on Jetson Nano.

All these networks were also tested on Raspberry Pi using TensorFlow, TensorFlow Lite and Arm NN, which showed consistent results on every platform and framework. These baseline tests were compared against the TensorRT library on Jetson Nano platform.

The original CNN presents two output neurons that produces wrong output data on both of them. To discard any issue resulting of processing a multiple output neuron neural network, a single output net was tested. Figure 57 depicts the network architecture and also a wrong output value when compared with the baseline results. The conclusion of this single test suggests that processing two output neurons is not the cause of the wrong prediction.

Next, a simpler neural network was defined (Figure 58), basically consisting of an input layer that diverges into two output neurons. TensorRT could correctly predict the right output value, producing the same as predicted on the other frameworks. Since this layer can properly compute the two right values, the next created neural networks came up by extending the previous one by adding more layers.

Figure 59 defines the next extension of the previous neural network architecture with an additional convolutional layer. Unexpectedly, the evaluation of this network shows wrong output values again. The only difference introduced on the neural network was a convolutional layer.

To test if the wrong predicted values are coming due to the presence of convolutional layers, two additional neural networks were created, which are not composed by any convolutional layer.

Figures 60 and 61 show the last versions of the original CNN. Figure 61 has the particularity of carrying all the original layers except the convolutions. Tests showed TensorRT could successfully predict the correct output values using both neural networks, which indicates the issue is coming from the convolutional layer.

The performed tests shows the inference issue comes by the execution of any convolutional layer, which contradicts the TensorRT API caliming to support convolutional operations. The evaluation was originally performed on Jetson Nano, which at the time was setup with CUDA (version 10.2) and TensorRT (version 5). The latest version (TensorRT 6) was not available yet to embedded devices, only on desktop environments.

After the the development and evaluation period, NVidia released the updated TensorRT version 7 and support for the TensorRT version 6 on embedded devices. So a fresh installation

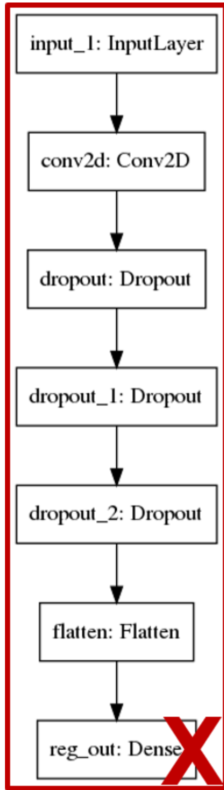


Figure 57: Single output neural net w/ a convolut layer

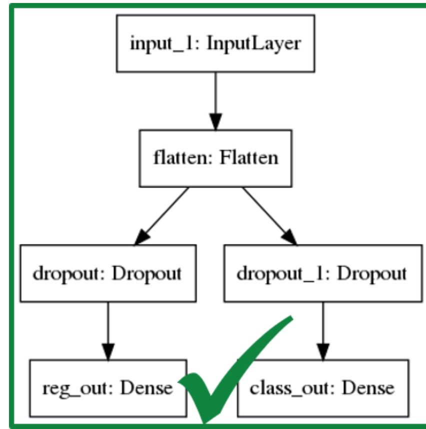


Figure 58: Basic neural net with two output neurons

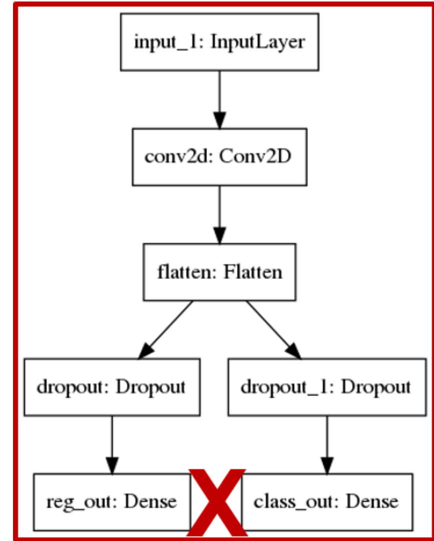


Figure 59: Single convolutional layer followed by two output neurons

of the full environment was performed, carrying the most recent packages and libraries. This operation was the last try to fix the wrong prediction coming from the convolutional layers.

After the full-setup with CUDA (version 10.2) and TensorRT (version 6), a last evaluation of the original CNN was done.

Now, the output results came correct and the issue presented in Figure 55 was no more occurring. Original errors were coming from internal issues on previous TensorRT versions, and upgrading all NVidia environment fixed the original inference output issue.

To detect traffic signs

Some challenges were also faced when fully exploiting GPU-cores of Jetson Nano to aid support to the algorithm processing. To enable this secondary processing unit, which manages to run DL workloads in a more efficient way than the CPU, TensorRT inference engine was used.

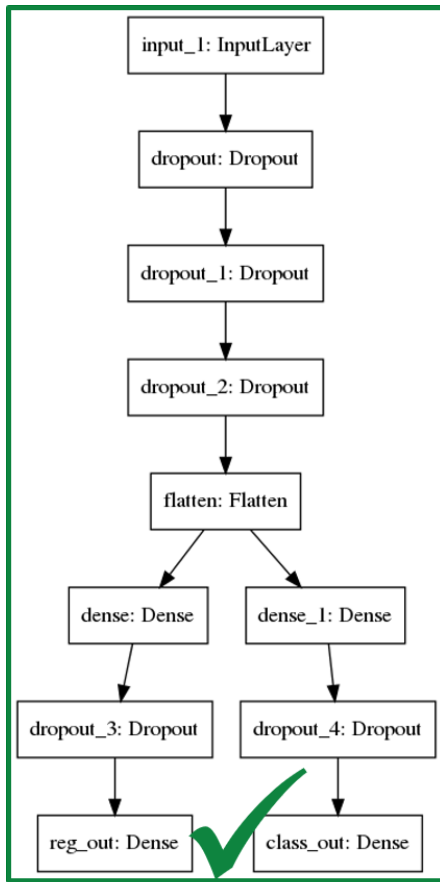


Figure 60: CNN without the convolutional and the max pooling layers

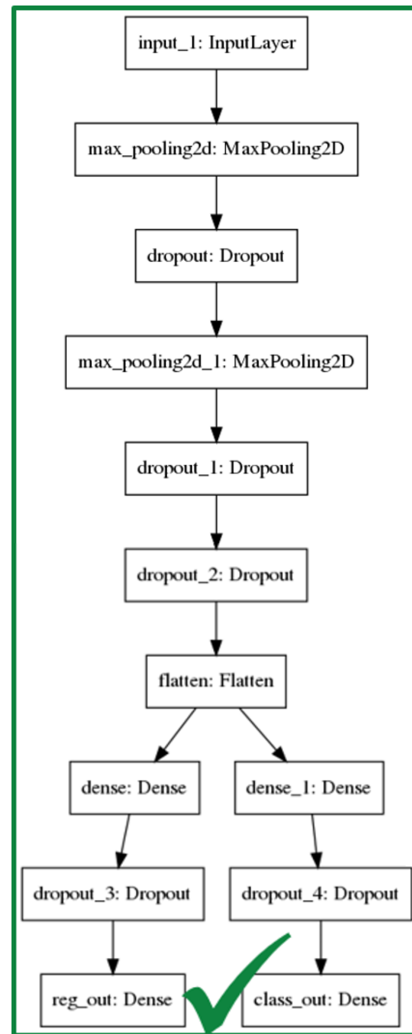


Figure 61: CNN without convolutional layers

TensorRT starts by importing the FgSegNet trained model and by allocating enough memory to save the predicted data, namely a matrix of the size of the original input image. On the communication mechanism level, the CPU maps the image to be readable by the GPU. The coprocessor executes the inference phase and maps the output data back to the address space that contains the original images, allocated by the CPU.

After all the necessary stages had been set to enable GPU processing, at runtime, the engine detected an unsupported operation on the imported model (Figure 62).

```
[TensorRT] ERROR: UffParser: Validator error: up_sampling2d_1_2/ResizeNearestNeighbor: Unsupported operation _ResizeNearest_TRT
[TensorRT] ERROR: Network must have at least one output
[TensorRT] ERROR: Network validation failed.
Traceback (most recent call last):
  File "trt_predict_mask.py", line 210, in <module>
    with builder.build_cuda_engine(network) as engine:
AttributeError: __enter__
```

Figure 62: TensorRT error while parsing the FgSegNet model

The problem resides on an operation — Resize Nearest Neighbor — that is present into the UpSampling layer and that is not natively supported by the TensorRT engine.

TensorRT allows the creation of custom layers inside the engine, so it can map the original one that is not supported by the NVidia engine into a customized one.

To overcome this parsing issue of the engine, a custom plugin was added that simulates the Resize Nearest Neighbor operation. The custom plugin node was registered in the TensorRT Plugin Registry and the namespace of the original unsupported operations were mapped into the customized ones, now available in the registry.

When importing the FgSegNet model, the graph was re-created and re-optimized by the engine, now using the plugin suporting the Resize Nearest Neighbor operation.

TensorRT used external preprocessing tools that allowed to transform the internal model graph, tuning it by replacing all the Resize Nearest Neighbor operations present on the UpSampling layer by the operations defined on the plugin.

After the preprocessing phase, the NVidia engine could successfully import the pre-trained model and properly execute the inference phase. Figure 63 shows one inference execution, segmenting a reasonable area around the real traffic sign presented on the input image.

Figure 63: FgSegNet prediction using TensorRT

4

EXPERIMENTAL RESULTS

After the development and the success deployment of the multiple implementations, the embedded systems that controls the RC vehicle were properly evaluated.

In this chapter, two types of experimental results are discussed: a quantitative and a qualitative assessment.

Both evaluations were taken using either the Raspberry Pi and the Jetson Nano, running on the different implementations with the same goal of drive autonomously the RC vehicle

The experimental results were retrieved by a real-time evaluation of the system that was coordinating the RC vehicle to autonomously drive on the predefined path, as illustrated in Figure 64. The real-time evaluation and the obtained results were also retrieved using the developed automated evaluation script.

Figure 64: Autonomous driving of the RC vehicle

4.1 HOMOGENEOUS VS. HETEROGENEOUS: A COMPARATIVE EVALUATION

4.1.1 *Following a Path*

Raspberry Pi 4

This first evaluation was performed on the Raspberry Pi 4 embedded platform. For this test, the fours implementations were executed, based on TensorFlow, TensorFlow Lite, Arm NN and PyArmNN libraries or engines.

Three different neural networks were also used: a simple neural network, a CNN without data augmentation and the same CNN with data augmentation.

Results in Figure 65 shows the inference times of the four different libraries running inference, where TensorFlow is the slowest one. This is due to TensorFlow does not fully exploit multiprocessing and hardware advanced features, neither is the optimal framework to be used to run on an ARM device.

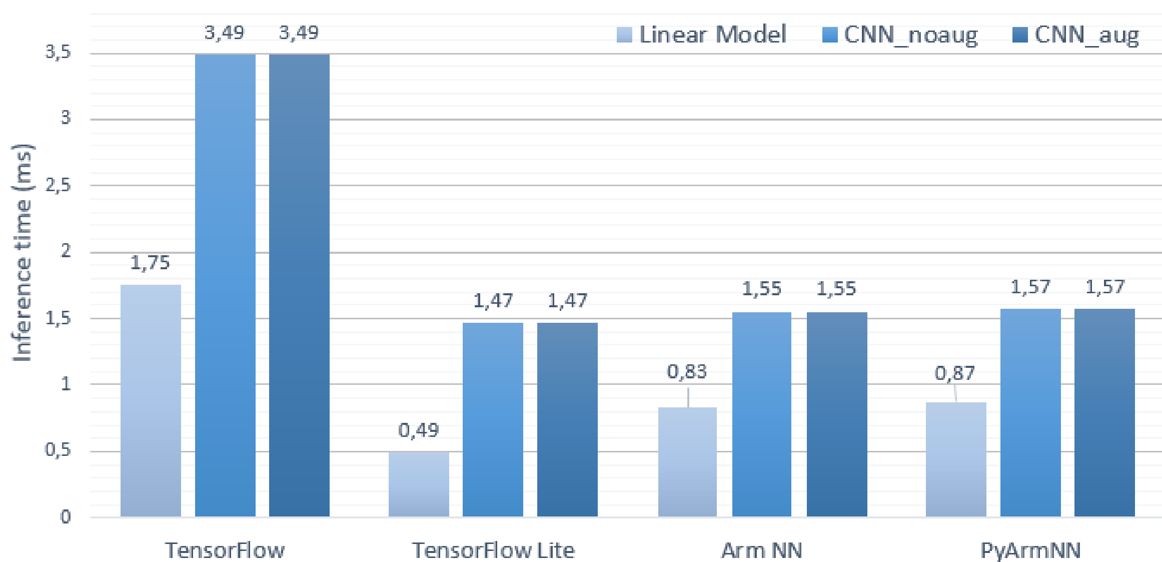


Figure 65: Inference times on Raspberry Pi 4

The TensorFlow Lite implementation can predict more frames per second, producing an output twice times faster than TensorFlow. The reason TensorFlow Lite can deliver more Frames Per Second (FPS) is because it is highly optimized for embedded devices. Leading to the appliance of several optimizations on the neural networks after had been internally loaded, such as pruning techniques, which eliminates irrelevant weights from the network.

Arm NN speed is quite similar to TensorFlow Lite execution, delivering approximately 645[1] FPS (against the 680 FPS of the TF Lite).

1 Value obtained by using CNN_noaug or CNN_aug model

The Arm NN inference library fully exploits the hardware capabilities, specially the ARM Cortex-A architecture. By using advanced optimization techniques alongside with optimizations introduced on the neural network model (as TensorFlow Lite), the obtained inference times are also more than twice faster comparing against the TensorFlow version.

In terms of inference time, PyArmNN implementation is roughly processing the same FPS, comparing against the Arm NN version. Execution times are identical because both uses the same Arm NN engine backend, the ARM Compute Library (ACL), which means both implementations are executing the same inference engine. The difference between the two implementations is the frontend interface. Inference phase of Arm NN implementation was developed using the C++ API, instead of PyArmNN that offers a tested Python interface environment that interacts with the Arm engine.

When comparing the different neural networks used, it is visible that the *Linear Model* (presented in Figure 65) is always faster predicting the data, despite the library or framework being used. Inference times are lower because the neural network is simpler than the CNN, holding less number of layers, which defines a smaller net. Consequently, this smaller neural network will have less matrix operations to be computed, running faster with a smaller amount of neurons to be computed.

However, running faster has a cost on the quality of the output predicted data, which will have less accuracy and a larger deviation from the baseline values of the predicted steering angle and on the efficiency to detect if the lane road is visible or not. This leads to a worst performance of the autonomous driving quality.

Although the *Linear Model* can provide higher FPS, making more decisions in real-time, the drawback comes when checking the quality of the predicted decisions. These decisions will not be as accurate as the predicted ones from both *CNN_noaug* or *CNN_aug*.

Test drives of the RC vehicle, naturally show a smoother driving and a bigger capacity of keep driving alongside the road lane, when using the CNN models instead of the *Linear Model*.

Technically, the CNNs presented in the previous figure are the same, carrying the network architecture. Their difference resides on the amount of data provided to the neural network at the moment of being trained.

CNN_noaug was trained with exactly the dataset generated while manual driving the vehicle.

CNN_aug is provided with the twice of the amount of images of the original dataset. Which means that it will be applied a data augmentation technique to double the data from the dataset. The data augmentation horizontally flips each image in the dataset, feeding the neural network with knowledge of steering the vehicle to the opposite way that it was driven.

A higher amount of data will impact the times of only the training phase of the neural network, which will, naturally, take more time to evaluate the additional data and to train

the convolutional neural network. However the time impact on inference phase will be none since both trained neural networks are equivalent, carrying exactly the same layers and neurons.

Since both CNNs are architecturally identical, results in Figure 65, shows exactly the same average inference time when using these models, despite the inference framework being used. As these two neural networks have an identical behaviour on inference phase and because *Linear Model* offers a poor accuracy of predicated values, only one CNN model will be tested on the following evaluations.

Memory Consumption

Followed by the execution times, the second important performance metric that must be evaluated is memory consumption. The resources of each embedded device must be analysed to check if there are some memory constraints causing bottlenecks on the implementations.

Results of Figure 66 shows that RAM usage is not a problem for the 4 GiB RAM in Raspberry Pi 4 despite the implementation being used.

The profiling indicates that the TensorFlow implementation uses more memory resources, up to a maximum of 125 MiB. The amount of memory used by the implementation reflects on a 3% of total RAM, still remaining about 2 GiB of free memory.

These results show memory usage is not a problem because the embedded system has enough available RAM when executing each implementation on Raspberry Pi 4.

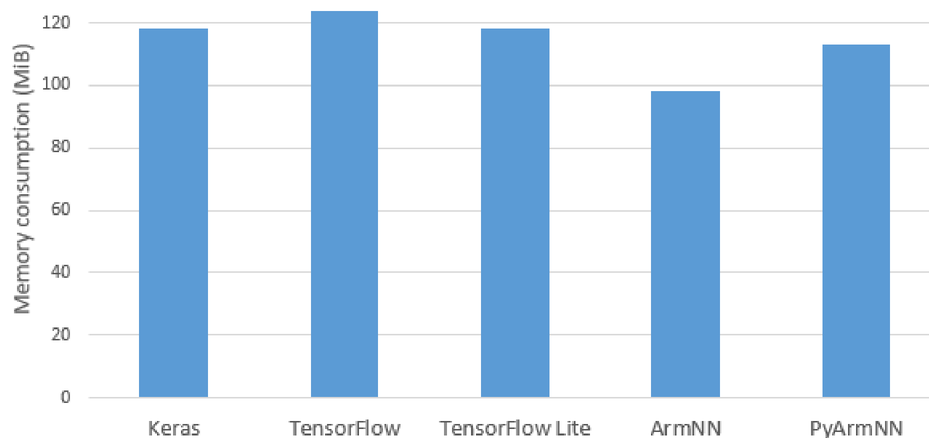


Figure 66: RAM Memory consumption on Raspberry Pi 4

Jetson Nano

Jetson Nano evaluation is described next, which used the *CNN_aug* model during the execution of the model. For this evaluation it were used the same metrics defined earlier for the Raspberry Pi: inference time (real-time metrics) and memory consumption.

An additional evaluation was performed on this system, it was performed a comparison between half-precision and single-precision floating-point execution times, since the GPU of this platform has native support for FP16 operations.

Multiple tests were executed on TensorRT to get the real values of the inference phase executed on the Jetson Nano accelerator. As described on the earlier chapter, TensorRT implementation requires additional preprocessing and memory maps, so that all data can be prepared and sent to the GPU.

There are three necessary stages needed to be executed to enable TensorRT inference engine: (i) the preprocessing, which transposes some matrices (input image) since TensorRT reads images differently[2] than the Raspberry Pi frameworks; (ii) the memory mapping, so the data saved on shared memory system can be readable also by the GPU; and (iii) the prediction itself.

Figure 67 presents the times obtained during each stage and inference times while executed on FP16 or FP32 mode.

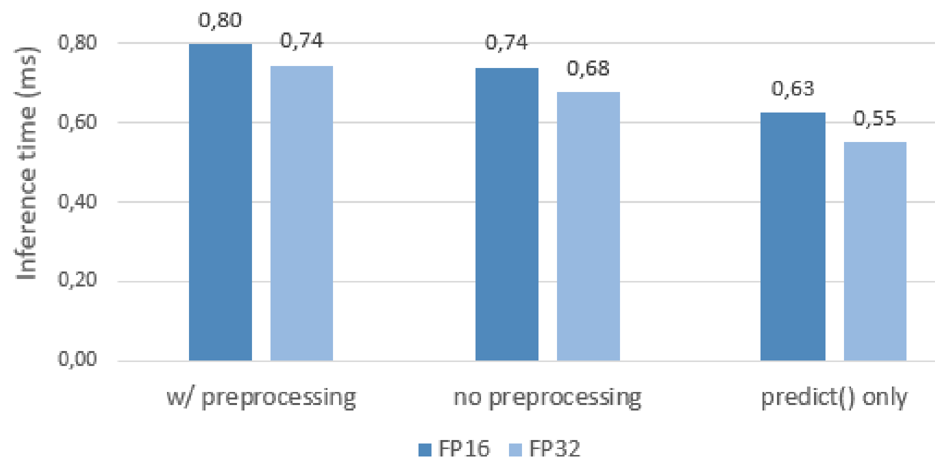


Figure 67: Inference times on TensorRT (half-precision vs. single-precision)

Analysing single-precision operations first, TensorRT inference takes on average, approximately, 0.55 milliseconds. Yet, considering only this operation on inference phase is not fair, since it is necessary to also perform, per every frame processed in real-time, additional computations. Thus to properly perform the prediction, the 0.55 ms needs to be added up to 0.68 ms to include a second stage — memory mapping. However, when taking in account preprocessing phase, the average time of total inference phase rises up to 0.74 milliseconds to frame.

Therefore, when analysing inference phase of TensorRT, mapping and preprocessing phase will be included on the overall inference phase, since they are required on every frame prediction.

2 Tensor inputs must be on the NCHW format instead of the conventional NHWC

Jetson Nano GPU has native support to perform half-precision operations, which theoretically would lead to doubling the performance, when comparing with single-precision execution. However, the times displayed in Figure 67 shows that the inference execution on FP32 mode performs faster than on FP16.

The developed code was validated to discard any potential issue during its development, but the following execution measurements showed that the half-precision mode kept executing slower.

The CNN model being used was originally trained to predict 64x48 frames, which is the image resolution being sent to the GPU. However a smaller resolution of this type can, eventually, perform slower when executing in half-precision because of the additional overhead introduced. Each value of every image is stored in memory with a single-precision floating-point format. Before half-precision processing, these values must be converted to a half-precision floating-point format. After all internal operations, output data must be converted again to single-precision format.

Because frames are too small, the necessary time to execute the precision-format conversion pipeline can overcome the advantage of the parallel half-precision operations, leading to a lack of performance when comparing with single-precision inference execution.

To fully exploit the FP16 capabilities of the GPU, a new CNN was trained. This new model it is defined by the same layers defined before, but instead of being trained to predict 64x48 images, the resolution was increased four times. Resulting on a model that theoretically needs more system resources to predict frames with resolution of 256x192 pixels.

An evaluation of this bigger model was executed, which resulted on performance improvements when executing prediction on half-precision mode, as presented in Figure 68.

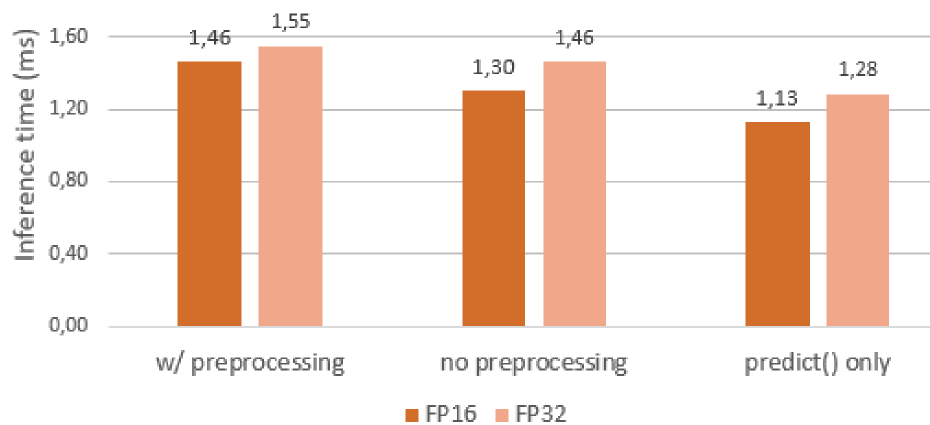


Figure 68: Inference times on TensorRT (half-precision vs. single-precision), using 256x192 image resolution

By increasing the image resolution, TensorRT executed inference mode faster on the FP16 format when compared against the FP32 format, with a speedup of approximately 10%.

The last evaluation shows that the half-precision mode is substantial faster, hence this is the preferable mode to be used when using large models. However, as the CNN being used on this use-case predicts small resolution images, greater performance is achieved using single-precision mode.

Memory Consumption

During the runtime of the TensorRT algorithm, a profiling was also performed. This procedure was driven to check if memory resources of Jetson Nano are enough for the real-time algorithm developed.

Memory tests are visible in Figure 69. It is noticeable the algorithm uses a maximum of 490 MiB when using the simple *Linear Model*, and a maximum of 1004 MiB when using the CNN intended to be deployed.

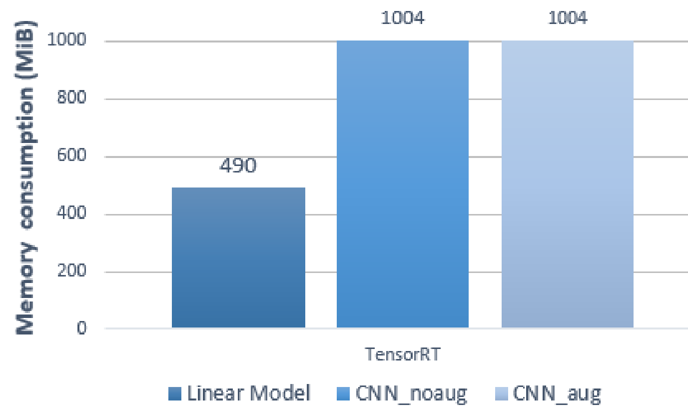


Figure 69: RAM Memory consumption on Jetson Nano.

This amount of RAM memory being used will not have a noticeable impact on the algorithm's performance since Jetson Nano has a total of 4 GiB of RAM. The OS normally uses approximately 1 GiB, on which it stills remains about 2 GiB of available memory.

The memory usage of the inference algorithm was handled by executing a kernel instruction to define an upper memory limit to be used by TensorRT algorithm. A performance impact happens when this limit is changed, as illustrated in Figure 70.

By reducing the upper-memory limit, inference times naturally decreased but maintained a good quality on the inference times, for the memory limitations that was confronted against. The minimum memory being used by TensorRT was about 100 MiB and yet inference time only decreased about 0.05 milliseconds.

For a maximum memory limit greater than 1000 MiB the performance stabilized, not existing any improvements on performance when TensorRT used more RAM.

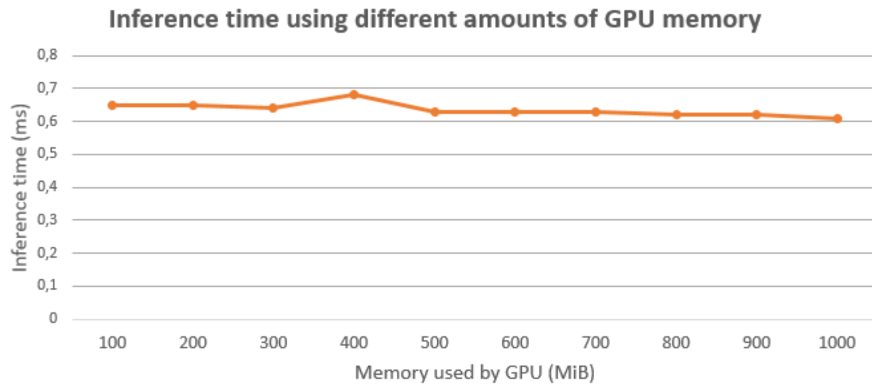


Figure 70: TensorRT inference executions varying the upper-limit of memory usage.

Comparative Evaluation

When comparing real-time inference execution times of both platforms, presented in Figure 71, it is possible to notice that the Jetson Nano implementation performs faster than all Raspberry Pi inference engines or libraries used.

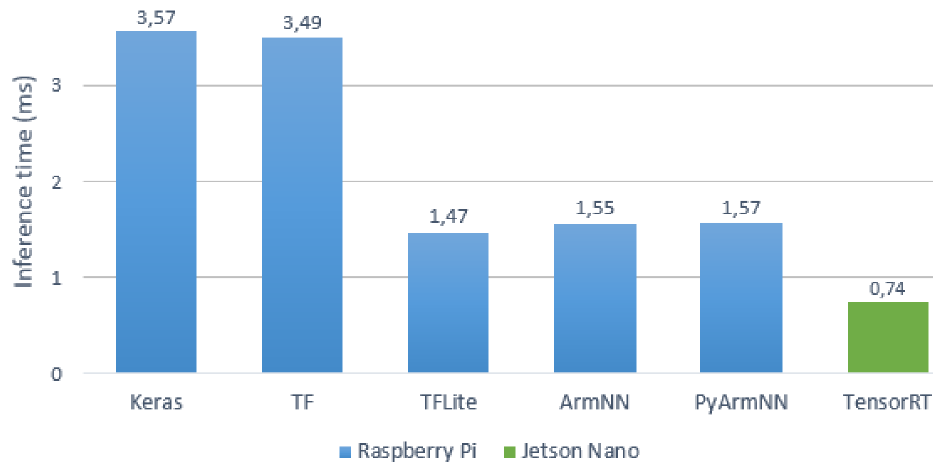


Figure 71: Overall comparison of inference times among both embedded systems

Tensorflow Lite algorithm is the fastest implementation on the Raspberry Pi 4, performing more than twice faster against the slowest algorithm (on this platform). This is due to the best use of TensorFlow Lite on advanced features of the Cortex-A CPU, such as enabling multiprocessing and the SIMD accelerator instructions, like described earlier.

On the other side, TensorRT, using the GPU-cores, performs each frame prediction even faster, in approximately 0.74 milliseconds — almost twice faster than the fastest Raspberry Pi implementation. This performance achievement is reached because Jetson Nano enables the GPU-cores to process the heavy DL workload.

As described on the section about Jetson Nano technical details, these hardware components can process more FLOPS, turning this embedded system more computationally powerful than Raspberry Pi 4 by empowering a co-processor. This unit will extend the main CPU to perform tasks that CPU cannot perform efficiently. Consequently, TensorRT inference execution can process almost the double of FPS, comparing against the ones processed by TensorFlow Lite on Raspberry Pi.

On both platforms, memory consumption is not a problem, since both embedded systems have 4 GiB of RAM, which is not fully used by the multiple implementations developed.

All Raspberry Pi implementations empower a lower memory usage, about an average of 110 MiB per implementation execution (Figure 66), against the 1 GiB on Jetson Nano. Yet, if there were memory limitations on the Jetson Nano, the memory manipulation would bypass this constraint. This constraint operation would lead to a minimal loss of performance on Jetson Nano. However TensorRT performance would keep to be greater than all Raspberry Pi implementations, despite of limiting memory to the same amount of 110 MiB.

It is not fair to compare only the inference times obtained on both embedded system against the driver reaction times that were studied on the state of the art chapter, because per each frame being processed some additional time must be accounted for. This is, not only the inference phase takes CPU or GPU time. The additional phases — preprocessing and communication phases — and synchronization mechanisms takes some additional time when executing these DL workloads.

A complete iteration time was measured, meaning the amount of time needed to capture the frame, preprocess it, synchronize all necessary routines, predict the data and send the information to the microcontroller was taken into account.

The complete execution time required to process each frame, using the multiple implementations on both embedded systems is displayed in Figure 72.

Real-time metrics

Earlier studies indicates that processing an image in real-time should not take longer than 300 milliseconds to simulate an human driver reaction time. Providing this maximum time for the frame to be processed and to the information be sent to the vehicle, mimics an ideal human driving reaction.

The camera installed on the RC vehicle was configured to capture images on a framerate of 30 FPS, which means that the time to process each frame must not exceed 33,33 milliseconds. If the resulting time did not stay within the defined range, a delay on driving reaction will occur. Meaning the frame being processed at the moment could had be took way too time, and the vehicle will be driving some distance ahead of the processing frame, resulting on outdated predicted information.

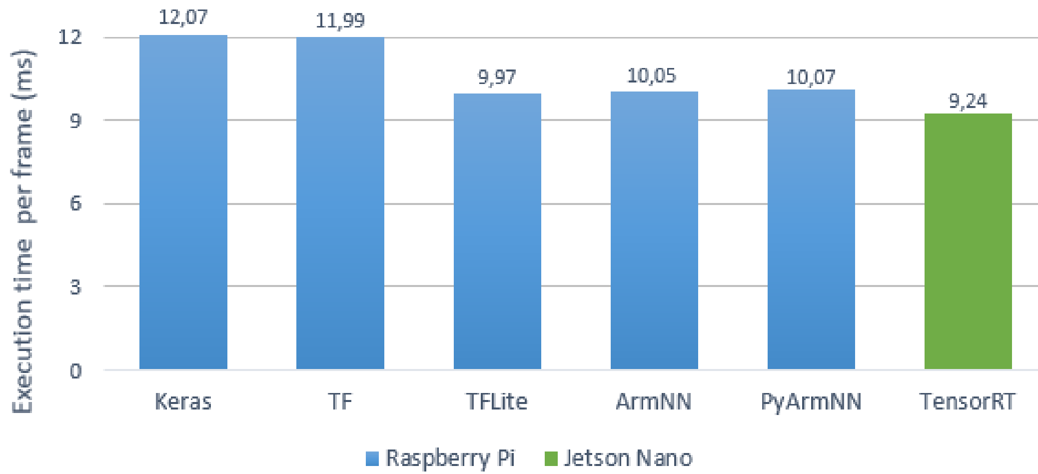


Figure 72: Overall comparison of execution times among both embedded systems

This necessary frame processing execution time is lower than the maximum 300 milliseconds of human driver reaction. This real-time processing time kept within the defined range, which indicates to react faster than the human drivers average.

The iteration time evaluation represented in Figure 72, reveals that the complete iteration time on Raspberry Pi 4 and Jetson Nano is taking approximately 10 ms and 9 ms, respectively. The actual DL workloads are performing comfortably well within the defined margin, which permits some complexity be added to the algorithm.

Giving the efficiency of the algorithms there was the necessity to increase the model complexity, since these workloads does not push enough the available system resources.

4.1.2 *Inference Performance of Traffic Signs*

A quantitative analysis of the traffic signs detection algorithm was executed to trace its performance during inference phase — the main aspect this dissertation aims to evaluate and to improve.

The algorithm was real-time again executed using different libraries — TensorFlow, TensorFlow Lite, Arm NN, PyArmNN and TensorRT — and the both embedded systems. Figure 77 presents an overall evaluation of inference times of the traffic signs algorithm.

Among the multiple developed implementations for Raspberry Pi, the one deployed using TensorFlow library was the fastest one, yet taking more than 6 long seconds to predict one single image. On the other hand, TensorFlow Lite is the slowest implementation.

It was expected TensorFlow Lite version could provide an higher throughput but this was not the case. Unlike the previous CNN model, TensorFlow could handle the DL model faster

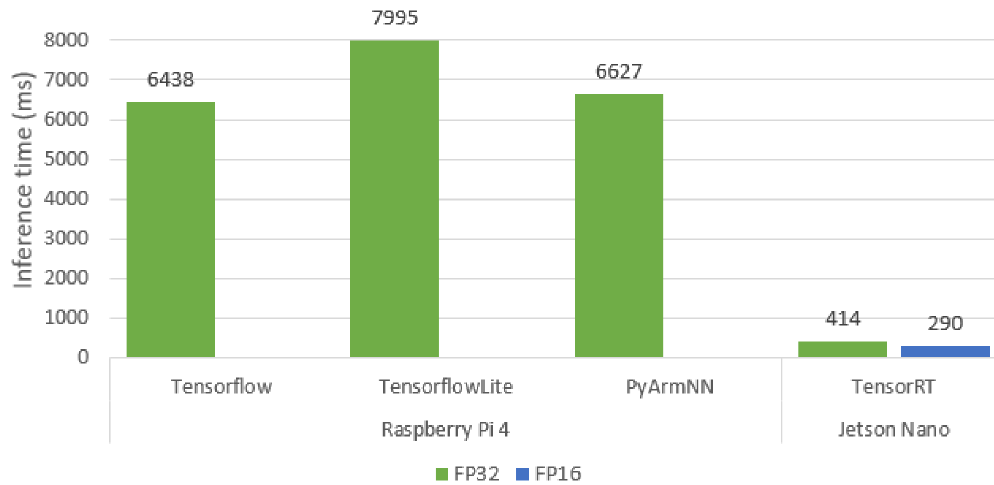


Figure 73: Overall evaluation of traffic signs inference on multiple libraries

than Lite version. The situation is due to the complexity of the FgSegNet model, presenting an higher number and variation of layers rather than the CNN.

This evaluation among both TensorFlow versions suggests the TensorFlow Lite library performs efficiently when parsing and optimizing light DL models. Where TensorFlow library is more suitable to run larger and complex models.

However, it is not recommended to run these Raspberry Pi implementations because it cannot provide the desirable FPS to properly identify a traffic sign in real-time.

Raspberry Pi 4, should not be used to run this specific workload since it takes more than 6 seconds to detect an eventual traffic sign on the road. Thus, using this embedded device to process the required use-case would lead to a major delay on detecting traffic signs. The RC vehicle would not be capable to stop on an eventual stop sign, which would be detected only 6 seconds later.

Nevertheless, there is a major improvement on the efficiency of the algorithm while being executed on TensorRT. This heterogeneous implementation enables GPU processing of the Jetson Nano embedded device, improving much more the inference comparing with the Raspberry Pi implementations. The optimizations carried by TensorRT might lead to a speed-up up to 27 times, comparing with TensorFlow (on Raspberry Pi).

These inference times carried by Jetson Nano are quite suitable to perform the required real-time tasks without major delays on the detection of traffic signs. TensorRT can predict a traffic sign from an image in about 414 milliseconds, which summing with the remaining pre and post-processing operations, does not reach the maximum suggested of 700 ms per iteration. This efficient real-time processing, marks the Jetson Nano platform as ideal (from both compared embedded devices) for this complex DL workload processing.

Half-precision

As explored on the CNN model, Jetson Nano is the only platform that allows to exploit half-precision operations during runtime. TensorRT also allows to enable such advanced optimization to improve the performance, depending of the algorithms.

During the FgSegNet model execution, this mentioned feature was also explored, which led to the results presented in Figure 74.

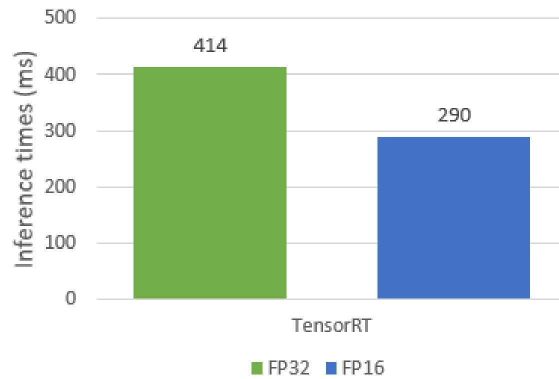


Figure 74: Traffic signs inference time on TensorRT using half and single-precision

It took about 410 ms to detect a traffic sign from an image, on FP32 mode. Yet, this time can be decreased if half-precision mode was enabled. Floating-point manipulation evaluations indicates the algorithm runs approximately 25% faster on half-precision, comparing against single-precision execution.

An ideal speedup of two times is not reached, when enabling half-precision, because there are multiple additional data conversions when this mode is activated.

When running on this particular mode, the stored weights of FgSegNet model are converted from a representation of 32-bits into 16-bits. The input frame being predicted suffers the same conversion, as well the output image which comes on a 16-bits representation. Output image is internally converted again into a 32-bits representation, so the data can be used. These additional steps do not contribute to an ideal half in inference time, comparing against single-precision mode. Yet, this additional pipeline and half-precision mode execution can process faster than single-precision.

Memory Consumption

Memory consumption of the the multiple implementations was also evaluated, which is presented in Figure 75.

The initial Raspberry Pi 4 (2 GiB RAM version) deployment issue revealed the TensorFlow could not run FgSegNet because the algorithm was consuming more than 1 GiB of the system resources. As the device was using about 1 GiB to OS management, the algorithm could not

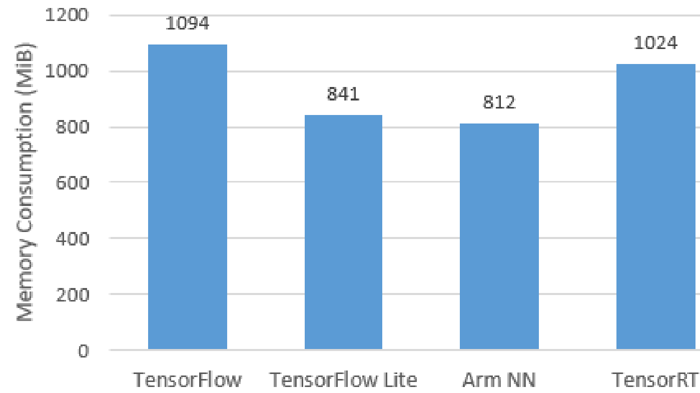


Figure 75: Memory consumption of traffic signs detection algorithm on multiple libraries

properly execute. Therefore it is necessary a platform with a minimum of 4 GiB of RAM to properly execute this particular version.

The two remaining libraries — TensorFlow Lite and Arm NN — used about 800 MiB of memory to run the DL workload, and therefore the 2 GiB Raspberry Pi 4 version is enough.

Thus, the original Raspberry Pi version (with 4 GiB of RAM) to target these DL workloads do not present memory constraints since the maximum memory used by these implementations is about 1 GiB.

On the other system, the algorithm developed for the Jetson Nano initially used about 1024 GiB of memory. This rounded value was achieved because it is the default maximum value of memory that the processes have access to. Therefore, the memory being used by TensorRT can be manipulated, and the user can set a maximum arbitrary value of memory being accessed by TensorRT engine.

4.1.3 *Traffic Signs: Number of Layers in the Neural Net*

An extensive deployment process was required to correctly and efficiently run the different DL models on both embedded devices. So it is important, from a deployment point of view, to profile the given machine learning models, at a layer level, to find possible performance issues or relevant performance data from layers that could lead to the construction of an efficient neural network.

If a machine learning expert knows the performance impact of each layer, it makes possible the creation of neural networks with inference performance in mind.

A profiling of the FgSegNet model was executed, where each layer was teased, and inference time per layer was obtained.

All the used inference libraries do not provide a direct mechanism to measure each layer inference time while predicting data. It is only possible to probe the first and last layer to measure the elapsed time between both, obtaining full inference time (of all layers).

To measure the elapsed time of each layer, a DL profiling tool was developed. After the tool import the model, several submodels are internally generated. The number of submodels created is equal to the number of layers the original model has. The generation of n submodels (where the original model has n layers) is dictated by the following rule: the first submodel has the first layer; the second submodel has the second layer; and so on until reach the n-th submodel defined by the n first layers.

Due to some constraints found during the creation of the tool, the obtained inference times per layer cannot be truly trusted, although an another relevant performance data was found.

Alternatively of what it can be believed on most times, the number of layers in a neural network have an impact on the final inference performance but not due to the increasing number. A trained model with an higher number of layers might be faster on inference phase than the same model with some layers that had been removed. This suggests most inference engines or libraries apply internal optimization of the imported model by merging or combining some layers.

Figure 76 presents the times of each submodel, where the submodel n is defined by the first n layers from the original model.

It is possible to visualize that *submodel 16* runs faster than *submodel 15*. Despite both models have 15 common layers with the exact parameters and process the same input, the submodel that presents one additional layer performs faster because the inference engine used introduces advanced optimizations within the model layers resulting on improvements on the performance during the inference phase.

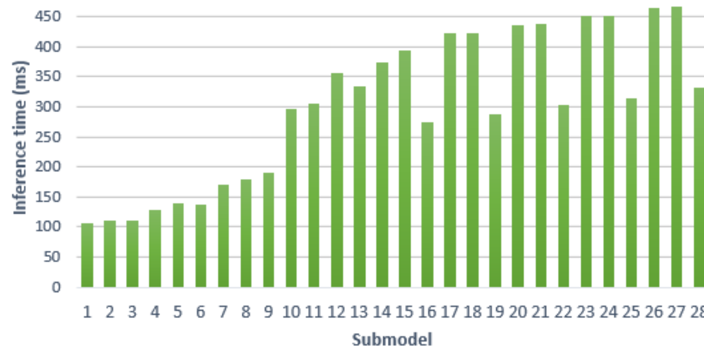


Figure 76: Inference time of generated submodels

4.1.4 *Traffic Signs: Latency and Throughput*

Latency and throughput metrics are quite important to compare the performance and to improve the efficiency of DL models.

Throughput is a measurement in ML that refers to the number of data units processed in one unit of time. The unit of this measure can be *images/ second*, where *image* is the input

data and *second* is the unit of time. More throughput means a better performance. This is an important measure when dealing with performance of non-real time applications.

On the other hand, latency is a measure that refers to the time spent to process one unit of data at a time. The latency is measured in time units, such as seconds or milliseconds. Less latency indicates a better performance and it is directly tied with the performance of real time applications or systems.

Higher throughput means the algorithm can process an higher amount of images, in parallel, per unity of time. Lower latency means the algorithm can process a single image faster, in a such period of time.

On this use-case, to achieve higher throughput levels on inference, multiple frames must be recorded first. When the set of frames is collected, this data is fed to the network, which processes them and predicts the data of each one. However, on this use-case, having a batch size higher than one, on inference, is not an advantage, since the whole batch will the processed in parallel but the RC vehicle vehicle needs the first frame predicted first, and so on, so the car can sequentially analyse the frames and follow the lane road.

On this real-time use-case a higher throughput will not be beneficial: although the engine is processing each image in a shorter time, it takes longer to generate the output of two or more predicted images than a single one. In this case, when increasing the throughput, the algorithm will take longer per iteration and the FPS will decrease.

The above case is not ideal because by increasing the throughput, the latency is worsen, which plays an important role on real-time processing. This use-case requires the minimal latency possible from the algorithm because the priority is to reduce the time spent processing one single image, because this real-time system requires each frame to be processed sequentially, generating faster output data.

4.2 QUALITATIVE ASSESSMENT

This subsection presents a qualitative analysis of the RC vehicle driving mode.

Previous results shows that both embedded systems were efficient using the convolutional neural network, which is capable to predict critic data to the vehicle follow a predefined path.

When adding up the traffic sign detection feature, Raspberry Pi 4 took too many seconds to predict output data from each frame. However, Jetson Nano could easily predict the steering angle in milliseconds only, which was a requirement to real-time processing.

These results does not directly indicate the car is able to autonomously drive on the path by strictly following the lane, either curved or straight. Which means the earlier positive results might not reflect on an ideal autonomous driving operation, namely the capability of the RC vehicle to follow the lane, steer the wheels properly and not to deviate its direction from the lane orientation.

This qualitative analysis and evaluation will be presented in two subsections. On the first one it is presented the evaluation of the driving itself, later is presented a qualitative analysis of the traffic signs detection.

4.2.1 *Raspberry Pi vs. Jetson Nano to Follow a Path*

In order to the RC vehicle perform accurately whilst accelerating through the path and steering towards the orientation of the road lane, an optimal training data is required. This means that the vehicle will have a smoother performance steering the wheels as accurate is the training data, which is used to train the CNN.

Despite the training data had to be collected manually by manually driving the vehicle, and therefore it is not the most reliable data existing, both embedded systems had a good behaviour considering its autonomous driving mode.

Both the Raspberry Pi 4 and the NVidia Jetson Nano systems could capture, process and predict data within a small range of time that matches the required one to compute in real-time. Also the accuracy of the predicted data was higher. Both phenomenons leads to a smooth autonomous driving mode, hence both versions of the RC vehicle — on the Raspberry Pi and Jetson Nano — managed to successfully follow the predefined path without any impediments or anomalies.

However, to achieve this smooth driving mode a change had to be done on the settings of the vehicle speed controller. Initially, the vehicle could not follow smoothly the path, leading to frequent crashes. This phenomenon happened using the both embedded systems and it was found out this miss-behaviour was being generated by the initial car speed whilst following the road lane. A speed limitation through software was built, which fixed the problem of the vehicle of getting out of its route.

The maximum speed of the vehicle was lowered and thus the vehicle could properly steer correctly according the orientation of the road lanes.

The initial car crashes was also due to the non-optimal training dataset that did not possessed the most accurate information regarding to the steering angle direction being applied upon the lane view.

Autonomous driving with traffic signs

Due to the high performance achieved by processing the CNN responsible for the acceleration and steering of the vehicle, an additional workload was added: traffic sign detection.

The Raspberry Pi 4 could not handle both workloads. As the algorithm responsible for detecting traffic signs was taking too long — roughly 6 seconds per frame — there was not

possible to have ideal conditions to react in time to traffic signs. So the vehicle would always not react to any traffic sign in time, ignoring them during those six seconds.

On the other hand, using NVidia Jetson Nano, a required performance to real-time processing was achieved and on this use-case the vehicle could follow the predefined path smoothly, spinning the wheels according to the lane orientation, and also process the traffic signs algorithm.

However, when approaching a stop sign, the vehicle did not behaved as expected, not stopping before the stop sign. Jetson Nano could handle an autonomous driving but could not detect traffic signs, hence it always ended running over them. This was due to a issue regarding to the training of the model capable to detect the traffic signs that is explained next.

4.2.2 *Detection of Traffic Signs*

Previous sections presented multiple examples of successfully predicted traffic signs with good accuracy on the outcome of the segmented zones, yet this led to a misunderstood of how well can this model behave on different environments.

All tests of traffic signs detection were performed under a controlled environment, so the test-images from Cityscapes dataset used to train the model are quite different to the ones captured from the RC vehicle in real time.

When detecting traffic signs with similar images to the Cityscapes dataset, the FgSegNet application could easily identify multiple traffic signs in the input image. However, an opposite behaviour, with bad accuracy, began to be registered when using images of an uncontrolled environment — the ones being recorded by the scaled vehicle.

Several isolated tests were executed, which consisted on predicting traffic signs from images that does not contains a city environment, unlike the Cityscapes dataset, not being taken under a controller environment.

Results presented in Figure 77 shows that the model could not properly identify traffic signs when these objects are outside of a road or a city environment. This particular behaviour happens due to the restricted trained environment provided by the Cityscapes.

As the FgSegNet was trained using this specific dataset, which consists of thousands of images recorded in multiples cities, the model will naturally learn how to detect traffic signs based in these specific conditions only.

The RC vehicle is naturally being tested on a uncontrolled environment, very different to the Cityscapes environment. The track built to the vehicle drive was built inside a building, in a laboratory, which does not resemble to a city environment at all, like illustrated in Figure 78 and Figure 79.

Mini-scaled stop signs were placed on the track, and when the vehicle was following it, it struggle to detect the sign ahead and sometimes it did not stopped. Consequently, a random

Figure 77: Prediction of an uncontrolled environment images using FgSegNet

Figure 78: Environment of predefined track of RC vehicle

Figure 79: Example of a frame recorded by the RC vehicle

red marker was also placed on the path, and as the model can run with very poor accuracy, sometimes the marker was identified as a traffic sign, which lead that the vehicle has stopped wrongly multiple times.

5

CONCLUSION

This dissertation aimed to improve the inference performance of DL algorithms on embedded systems. To test and evaluate this type of workloads, a testbed environment was used in this project: a prototype of a reduced scale remote controlled vehicle and two different embedded systems. The goal of the algorithms was to autonomously drive the RC vehicle. A real-time process is always capturing images from a camera on top of the vehicle. Each image is analysed by the inference engine which predicts the steering angle to be directly sent to the wheels through a microcontroller.

The tested embedded systems used were a Raspberry Pi, whith a 4-core ARM Cortex-A72 device that targets multiprocessing, and a NVidia Jetson Nano, a heterogeneous system with a 4-core ARM Cortex-A57 device (a previous architecture generation) and 128 additional Maxwell CUDA-cores on its NVidia GPU.

To achieve a better performance, the algorithms were deployed and evaluated into different frameworks and machine learning libraries. On Raspberry Pi 4, TensorFlow, TensorFlow Lite, Arm NN and PyArmNN libraries/inference engines were used to enable advanced ARM-based operations and optimizations. TensorRT library was used to engage the 128 CUDA-cores of Jetson Nano while executing the inference phase of the DL algorithms.

While deploying the algorithms on both embedded systems, multiple features were enabled to maximize performance. DL models suffered multiple conversions and optimizations that relied on the tools used that were mentioned before. Models were trained on single and half-precision mode, quantized, predicted on both floating-point modes, and took additional performance techniques and approaches to optimize the inference phase execution time.

5.1 HOMOGENEOUS VS. HETEROGENEOUS: A COMPARATIVE EVALUATION

An extensive debugging phase and bug fixes was performed to correctly install the tools, deploy and execute the algorithms on the supplied embedded systems and to achieve a reasonable real-time performance, so the vehicle could autonomously drive with low-latency levels.

Initially, multiple incompatibility errors on multiple libraries and packages were found due to the lack of support of the available software versions that the algorithms were deployed into.

On deployment, major performance issues were faced and fixed, such as the hardness to engage multiprocessing of ARM CPU, when using Arm NN library, and the difficulty to run the half-precision inference model on Jetson Nano, using TensorRT.

The performed evaluations on Raspberry Pi 4 showed a speedup higher than two, when using any tested framework or library, except for the TensorFLow. TensorFlow implementation was the slowest one, compared against the mentioned Raspberry Pi tested tools, having an inference time more than twice slower than the remaining tools. TensorFlow Lite was the fastest implementation, delivering approximately 645 FPS. Memory consumption did not reveal to be a problem because none implementation exceeded 150 MiB of memory usage. The Raspberry Pi 4 has 4 GiB of memory RAM.

When using Jetson Nano as the controller embedded system, the performance improved even more because inference computations were handled by the GPU-cores.

Comparing against the TensorFlow Lite implementation — the fastest one of the Raspberry Pi 4 — TensorRT speedup was almost two.

However, when enabling FP32 mode, the performance had decreased. The DL model — a CNN — used on inference does not present a high number of layers and the complexity of their operations is not high, leading to a drawback when using half-precision on this particular CNN.

As the DL algorithm was not using a quite complex CNN, both embedded systems could handle it without major limitations. Consequently, the RC vehicle vehicle could be autonomously driven without major delays and improper latency values, while the algorithm was being real-time processed.

None embedded system was pushed towards its computational power limits when executing the original algorithm that used the CNN. To understand the limitations of both systems that take control of the vehicle driving, a new algorithm was implemented.

An additional DL algorithm was developed that loads an additional and more complex neural network. The goal of the new algorithm is to detect traffic signs and to stop the vehicle if any sign is close to the RC vehicle.

Evaluation results indicates the latency provided by the Raspberry Pi 4 when computing this additional workload is extremely high. The fastest implementation was achieved on TensorFlow, but each image processing takes longer than 6 seconds, which is not feasible for real-time detecting traffic signs.

Although the homogeneous system could not handle this new DL workload, Jetson Nano managed to run it quite efficiently. TensorRT implementation got a speedup of more than 15 times, comparing against the fastest Raspberry Pi 4 implementation. Each image processing

takes approximately 414 ms, which is less than 700-1000 ms, the minimum range suitable for this kind of detection.

However, the qualitative retrieved data results had shown that the ability to detect traffic signs is weak. The model used on this algorithm cannot provide accurate results when detecting traffic signs, which leads the RC vehicle not to stop when a sign is present, or vice-versa.

5.2 FUTURE WORK

This dissertation aimed to improve and evaluate the performance of DL workloads that used multiple neural networks, which have been previously trained. The CNN responsible for the autonomous driving had to be trained through training data collected from manually driving the vehicle. The outcome results and the precision of this data depends on the user that its collecting them. The better the user drive the vehicle, the better it will perform on autonomous driving mode.

As the precision of the training data retrieved was not perfect and was affected by the user performance, it motivates the following suggestions for improvement: a virtual simulation to automatically train the vehicle in accurate conditions and not user dependent.

Also not restricting this project to the specified tools and embedded systems, there are some more possible enhancements:

- To upgrade the hardware capabilities of Raspberry Pi 4 by adding an Intel Neural Compute Stick;
- To improve the accuracy of the traffic signs detection;
- To improve and extend the DL model profiling;
- To explore additional ML libraries and frameworks.

BIBLIOGRAPHY

Herbertand Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Proceedings of the 9th European Conference on Computer Vision, Part I*, pages –. Springer, 2006.

Dan Ciregan, Ueli Meier, and Jurgen Schmidhuber. Multi-column deep neural networks for image classification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages –, 2012.

Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, pages –. IEEE, 2016.

Vinayak V. Dixit, Sai Chand, and Divya J. Nair. Autonomous vehicles: Disengagements, accidents and reaction times. *PLoS ONE*, 11(12):1–14, 2016.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. Technical report, 2016.

Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2nd edition, 2017.

Tullio Giuffrè, Salvatore Trubia, Antonino Canale, and Alessandro Severino. Automated Vehicle: a Review of Road Safety Implications as Driver of Change. *27th CARSP Conference*, 2017.

Alexander Goldenshluger and Assaf Zeevi. The Hough transform estimator. *Annals of Statistics*, 32(5):1908–1932, 2004. URL https://doi.org/10.1214/009053604000000760.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

Allison Gray, Chris Gottbrath, Ryan Olson, and Shashank Prasanna. Deploying Deep Neural Networks with NVIDIA TensorRT, 2017. URL https://developer.nvidia.com/blog/deploying-deep-learning-nvidia-tensorrt/.

Jack Greenhalgh and Majid Mirmehdi. Recognizing text-based traffic signs. *IEEE Transactions on Intelligent Transportation Systems*, 16(3):–, 2015.

Brendan Gregg. The Flame Graph, 2016. URL http://www.brendangregg.com/flamegraphs.html.

Song Han. *Efficient Methods And Hardware For Deep Learning*. PhD thesis, Stanford University, California, 2017.

Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Hoda Imam, Bassem A Abdullah, Hossam E Abd, and El Munim. Semantic Segmentation under Severe Imaging Conditions. In *Proceedings of the Digital Image Computing: Techniques and Applications (DICTA)*. IEEE, 2019.

Petros A. Ioannou and C. C. Chien. Autonomous intelligent cruise control. *IEEE Transactions on Vehicular Technology*, 42(4):–, 1993.

Irida Labs and Silexica. Optimizing Deep-Learning Inference for Embedded Devices (white paper). Technical report, Silexica, 2018.

Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient Deep Learning Inference on Edge Devices. In *Proceedings of the ACM Conference on Systems and Machine Learning (SysML'18)*, 2018.

Ebrahim Karami, Mohamed S Shehata, and Andrew J Smith. Image Identification Using SIFT Algorithm: Performance Analysis against Different Image Deformations. *CoRR*, 2017. URL http://arxiv.org/abs/1710.02728.

Alexandros Kouris, Stylianos I. Venieris, Michail Rizakis, and Christos Savvas Bouganis. Approximate LSTMs for Time-Constrained Inference: Enabling Fast Reaction in Self-Driving Cars. *IEEE Consumer Electronics Magazine*, 9(4):11–26, 2020.

Andy Lee. Comparing Deep Neural Networks and Traditional Vision Algorithms in Mobile Robotics. 2016.

Xunyi Li, Jinju Shao, Guo Wei, and Ruhong Hou. AEB Control Strategy and Collision Analysis Considering the Human-Vehicle-Road Environment. In *Pervasive Systems, Algorithms and Networks*, pages 335–346. Springer, 2019.

Long Ang Lim and Hacer Yalim Keles. Learning multi-scale features for foreground segmentation. *Pattern Analysis and Applications*, 23(6):–, 2019. URL https://doi.org/10.1007/s10044-019-00845-9.

Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN Model Inference on CPUs. In *Proceedings of the USENIX Annual Technical Conference*, page 16. The Advanced Computing Systems Association, 2019.

Hengliang Luo, Yi Yang, Bei Tong, Fuchao Wu, and Bin Fan. Traffic Sign Recognition Using a Multi-Task Convolutional Neural Network. *IEEE Transactions on Intelligent Transportation Systems*, 19(4):–, 2018.

Saturnino Maldonado-Bascón, Sergio Lafuente-Arroyo, Pedro Gil-Jiménez, Hilario Gómez-Moreno, and Francisco López-Ferreras. Road-sign detection and recognition based on support vector machines. *IEEE Transactions on Intelligent Transportation Systems*, 8(2):–, 2007.

Calonder Michaeland, Lepetit Vincentand, Christophand. Strecha, and Fua Pascaz. BRIEF: Binary Robust Independent Elementary Features. In Petros Daniilidis Kostas }and Maragos and Paragios Nikos, editors, *Proceedings of the Computer Vision – ECCV*, pages 778–792. Springer, 2010.

James Montantes. Deep Compression: Optimization Techniques for Inference & Efficiency, 2019. URL https://www.kdnuggets.com/2019/03/deep-compression-optimization-techniques-inference-efficiency.html.

Lin Ning and Xipeng Shen. Deep Reuse: Streamline CNN Inference On the Fly via Coarse-Grained Computation Reuse. In *Proceedings of the ACM International Conference on Supercomputing*, page 438–448. North Carolina State University, Association for Computing Machinery, 2019.

NVidia. Deep Learning Frameworks Documentation, 2018. URL https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html.

NVidia. NVidia TensorRT, 2019. URL https://developer.nvidia.com/tensorrt.

NVidia. Power Management for Jetson Nano and Jetson TX1 Devices, 2020. URL https://docs.nvidia.com/jetson/l4t/index.html#page/TegraLinuxDriverPackageDevelopmentGuide/power_management_nano.html.

Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep Learning vs. Traditional Computer Vision. In *Proceedings of the Computer Vision Conference (CVC)*, volume 943, pages 128–144, 2019.

Antonio Polino, Eth Zürich, Razvan Pascanu, Google Deepmind, and Dan Alistarh. Model compression via Distillation and Quantization. In *Proceedings of the International Conference on Learning Representations*, Vancouver, 2018.

Qing Qin, Jie Ren, Jialong Yu, Ling Gao, Hai Wang, Jie Zheng, Yansong Feng, Jianbin Fang, and Zheng Wang. To Compress, or Not to Compress: Characterizing Deep Learning Model Compression for Embedded Inference. In *Proceedings of the IEEE Intl Conf on Parallel &*

Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications, 2018.

Edward Rosten and Tom Drummond. Machine Learning for High-Speed Corner Detection. In Ales Leonardis, Horst Bischof, and Axel Pinz, editors, *Proceedings of the Computer Vision – ECCV*, pages 430–443. Springer, 2006.

Artem Savkin, Thomas Lapotre, Kevin Strauss, Uzair Akbar, and Federico Tombari. Adversarial Appearance Learning in Augmented Cityscapes for Pedestrian Recognition in Autonomous Driving. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3305–3311, 2020.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2011.

TensorFlow. Model optimization, 2020. URL https://www.tensorflow.org/lite/performance/model_optimization.

Frank C D Tsai. Geometric hashing with line features. *Pattern Recognition*, 27(3):–, 1994. URL http://www.sciencedirect.com/science/article/pii/0031320394901155.

TVM. Open Deep Learning Compiler Stack, 2016. URL https://github.com/apache/incubator-tvm.

Angela Wang. Parallelizing across multiple CPU/GPUs to speed up deep learning inference at the edge, 2019. URL https://aws.amazon.com/pt/blogs/machine-learning/parallelizing-across-multiple-cpu-gpus-to-speed-up-deep-learning-inference-at-the-edge/.

Bharath Zadeh, Reza Bosagh; Ramsundar. *TensorFlow for Deep Learning From Linear Regression to Reinforcement Learning*. O'Reilly Media, 2018.

Aston Zhang, C. Lipton Zachary, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Spring, 2020.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.