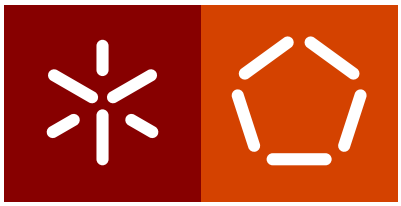


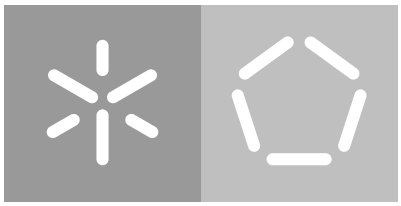
Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Pedro Alves Fernandes

Deeploy:
a Neural Network Computer Vision Tool

(for the NVidia Tegra TX2 Embedded System)

October 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Pedro Alves Fernandes

Deeploy:
a Neural Network Computer Vision Tool

(for the NVidia Tegra TX2 Embedded System)

Master Dissertation
Master Degree in Computer Science

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

October 2018

AGRADECIMENTOS

Esta dissertação não seria possível sem todos os que me ajudaram, direta ou indiretamente, na sua elaboração. A estes gostava de exprimir os meus agradecimentos.

Aos meus orientadores e coorientadores, Alberto Proença, André Pereira e André Ferreira que me auxiliaram, sempre com prontidão e dinâmica, ao longo do desenvolvimento deste projeto. Agradecer também à *Bosch* as excelentes condições de trabalho bem como a ajuda financeira (facultada através de uma bolsa).

Aos meus colegas de trabalho que me acompanharam de muito perto ao longo de todo o desenvolvimento desta dissertação e me auxiliaram, sempre que precisei, com ideias, sugestões ou palavras de incentivo.

Aos amigos que periodicamente me perguntava como estava o trabalho e que depois aguentavam os meus desabafos sobre o mesmo.

E por fim à minha família sem os quais, verdadeiramente, este trabalho não seria possível. Em especial, agradecer aos meus pais e ao meu irmão.

ABSTRACT

Machine Learning (ML) gives a computer system the ability to perform a certain task without being explicitly programmed to do it. Although ML is not a new topic in the field of computer science, these techniques have been gaining increasing popularity due to advances in hardware (especially GPUs). More powerful hardware supports more efficient training and a more responsive end-system, once deployed. These algorithms have proven to be particularly effective in image processing and feature detection, namely with deep neural networks.

In the context of a vehicle, autonomous or not, perceiving its external and internal environment enables the ability to detect and identify left behind objects, its misuse or other potentially dangerous situations. This captured data is relevant to trigger vehicle intelligent responses. *Bosch* is currently developing a system that has these capabilities and plans to leverage deep learning approaches to implement it.

This work aimed to test and evaluate the suitability of a given embedded device for the project. It also determined the best strategy to implement deep learning solutions in the device. The supplied test bed was a NVidia Software Development Kit (SDK) system for the embedded NVidia Jetson TX2 device with the System-on-Chip (SOC) Parker, an heterogeneous computing chip with 2 Denver-cores (a NVidia implementation of ARM-64 architecture), 4 CortexA57-cores (also ARM-64), 256 *Pascal* GPU-cores and support for up to 6 video cameras. The SDK includes several software library packages, including for image processing and ML.

With the goal of fully exploiting the embedded device compute capabilities, this work studied several inference frameworks, going as far as implementing an inference engine from scratch (named *Deeploy*) that produces inferences based on two libraries provided by NVidia: *cuDNN* and *TensorRT*. Deeploy was evaluated against well known and established frameworks, namely *Tensorflow*, *PyTorch* and *Darknet*, in terms of efficiency, resource management and overall ease of use, maintainability and flexibility. This work also exploited key performance related features available on the device, such as power modes, half-precision floating point computation and the implemented shared memory architecture between the GPU-cores and the CPU-cores.

RESUMO

Machine Learning dá a um sistema informático a capacidade de completar uma dada tarefa sem ser explicitamente programado para tal. Apesar de *Machine Learning* não ser um tópico novo no campo da engenharia informática, estas técnicas têm-se tornado cada vez mais comuns devido a avanços no *hardware* (especialmente nos GPUs). *Hardware* mais computacionalmente capaz dá origem a treinos mais eficientes e a sistemas em campo mais rápidos. Este tipo de técnicas, em especial redes neuronais, demonstraram-se eficazes no processamento de imagens e deteção de objetos.

No contexto de um veículo, autónomo ou não, perceber o seu interior e o ambiente no qual este se insere é essencial para detetar objetos esquecidos, o uso indevido do mesmo ou outro tipo de situações perigosas. Esta informação é essencial para desencadear respostas inteligentes por parte do veículo. A Bosch está atualmente a desenvolver um sistema com estas capacidades e para o implementar pretende utilizar soluções baseadas em redes neuronais.

Com o projeto pretendeu-se testar e avaliar a aptidão de um dado dispositivo embebido para este projeto. Serviu também para determinar a melhor estratégia para se fazer a implementação de redes neuronais neste dispositivo. Os testes foram feitos num kit de desenvolvimento da NVidia que consiste num NVidia Jetson TX2 que contém um *chip* de computação heterogéneo composto por 2 *cores* Denver (implementação da NVidia da arquitetura ARM-64), 4 *cores* CortexA57 (também ARM-64), 256 *cores* GPU Pascal e capacidade de se conectar até 6 camaras de vídeo. O kit de desenvolvimento inclui várias bibliotecas de *software* para processamento de imagem e até para *ML*.

Com o objectivo de tirar total partido das capacidades computacionais do sistema embebido, este trabalho explorou várias plataformas de inferência, implementando mesmo um motor de raiz capaz de fazer inferência recorrendo a duas bibliotecas desenvolvidas pela NVidia: cuDNN e tensorRT. Foi também feita uma comparação entre as duas implementações desenvolvidas e *frameworks* tradicionais como Tensorflow, PyTorch e Darknet no que toca a eficiência, facilidade de manutenção e flexibilidade. Este trabalho explorou também as *features* chave que estão relacionadas com *performance* disponibilizadas pelo dispositivo embebido, como modos de consumo de energia, computação numérica de virgula flutuante de meia precisão e a arquitetura de memória partilhada implementada entre os múltiplos *cores* ARM-64 e os CUDA-*cores* do GPU.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ANN Artificial Neural Network. 13, 15
API Application programming interface. 6
AVX Advanced Vector Extension. 11

BLAS Basic Linear Algebra Subprograms. 10, 30

CPU Central Processing Unit. ii, vi, 6, 7, 10, 11, 30, 32, 34–36, 57, 61–63
CUDA Compute Unified Device Architecture. 2, 3, 6, 8, 9, 11, 29, 30, 32, 34, 35, 45, 53

DNN Deep Neural Network. 12, 13, 37, 42

FMA Fused Multiply–Add. 9
FP16 Half-precision floating-point. 8, 20, 30, 50, 51
FP32 Single-precision floating-point. 8, 20, 50, 51
FP64 Double-precision floating-point. 8

GEMM General matrix multiply. 30, 34
GFLOPS Giga Floating-point Operations Per Second. 9, 10
GPU Graphics Processing Unit. ii, iii, vi, 2, 3, 5–11, 13, 29, 30, 32, 34–36, 39, 41, 53–55, 57–59, 61, 62
GUI Graphical user interface. 52

HPC High Performance Computing. 7

I/O Input/Output. 5, 6, 61
IEEE Institute of Electrical and Electronics Engineers. 8
IoU Intercept over Union. 27, 28
ISA Instruction Set Architecture. 3, 11

mAP Mean Average Precision. viii, 27, 28, 49, 50
ML Machine Learning. ii, iii, 12, 32, 35, 43

PCI-e Peripheral Component Interconnect Express. 7

RAM Random Access Memory. 11

SDK Software Development Kit. ii
SIFT Scale-invariant feature transform. 12

SOC System-on-Chip. ii
SOM System-on-Module. 5
SSE Streaming SIMD Extensions. 11

1

INTRODUCTION

Since the 30's the car has been able to self detect problems and warn the user of many mechanical problems Lamm (1984). The dreaded engine or ABS lights turn on once a problem with the corresponding systems has been detected, preventing possible dangerous situations. This is obviously possible due to an array of various sensors that monitor numerous aspects of the vehicle during its normal operation Baltusis (2004). All of the data generated by these sensors is analysed and if some values are out of the acceptable range the system alerts the user. This is the standard for the automotive industry today.

On the other hand, the interior of the vehicle is heavily neglected when it comes to self diagnostics. The most commonly available systems are mostly related to basic passenger safety (e.g., seat-belt reminder chime or automatic airbag deactivation of the passenger seat when minimum weight is not detected) ignoring passenger comfort, interior condition and more complex safety related subjects. Even the more advanced systems (more commonly found in higher-end models) focus mostly in the driver (e.g., drowsiness and distraction detection) Kaplan et al. (2015).

Bosch is currently developing a system to answer these needs. It should be able to oversee not only the driver (if one exists) but also all the vehicle occupants and cabin state. Its main goal is to detect miss-behaviour inside the cabin (e.g.: occupants fighting or poorly seated), left behind objects (e.g.: personal belongings or litter) and cabin damage. A system with these characteristics is appealing for car renting businesses but will play an even more important role in the self driving future.

Optimistically speaking, fully autonomous driving is expected to arrive around 2030 Litman (2017). Until then, a full cabin sensing system, like the one in development, is viable for car renting and sharing businesses. It should be capable of assuring that the vehicle is correctly and safely used and is acceptable for the next client. To a lesser extent, it can also prove to be interesting for taxi drivers by detecting miss behaviour and left behind personal belongings Brown (2018).

When fully autonomous vehicles arrive, a system like this one becomes even more relevant. Vehicle sharing is expected to become extremely common Gao et al. (2014); Lambert (2016) which creates the need to automatically protect both the vehicle and its passengers from potentially dangerous or damaging situations Broussard (2018). Tesla, one of the lead-

ing car manufactures researching self driving vehicles, already installed in its latest model (Tesla Model 3) a camera in its interior Lambert (2017). Although currently disabled, this camera effectively prepares their latest model for the car sharing future and enables the possibility of an analogous system being implemented with a software update.

Such a versatile and ambitious system can only be developed based on vision techniques. Its final version is expected to do the bulk of its monitoring tasks via an array of cameras installed in the cabin. This makes all visible events potentially detectable by the system and enables future functionality addition/tuning via a software update. One of the major drawbacks of such solution is the computational cost of real time image analysis. In particular, this analysis is expected to rely on state of the art deep learning vision models.

This work is part of the project described and will focus of the deployment and performance issues related with it. It is very performance oriented due to the high computation demand of the typical deep learning model. To answer these needs, the computing system selected was the NVidia Jetson TX2, an embedded device that has an integrated NVidia GPU with 256 Compute Unified Device Architecture (CUDA) cores, giving it an extremely high computational power when compared other devices of this category. Currently, the project does not have a deployment system selected, providing a very flexible environment for testing. On the other hand, the project already defined some deep learning architectures that will be used in the final product, namely *YoloV3*, making it the focus of this deployment effort.

1.1 Goals, challenges and contributions

The main goal of this dissertation is to explore the suitability of a given embedded device (NVidia Jetson TX2) to deploy a real-time vehicle interior sensing system stressing computer vision applications. The work focuses on testing different deployment strategies and multiple neural network models aiming to achieve the best possible performance. The deployment strategies tested contain a variety of different frameworks, from well known, high performance frameworks (e.g.: PyTorch and Tensorflow) to a custom tool developed specifically for this project named Deeploy. This tool is the main implementation effort of this work and will be one of its main focus.

A real time analysis of multiple camera feeds using neural networks is very computationally demanding Sze et al. (2017). For a single inference, a relatively well known classification network, *VGG-16* Simonyan and Zisserman (2014), performs 15.300.000.000 multiply/add float operations Howard et al. (2017). Thus, it becomes imperative to explore multiple deployment strategies and models to guarantee an efficient hardware optimisation.

The work in this dissertation aimed to reach the following goals:

- to evaluate the available Jetson TX2 performance features;

- to evaluate alternative inference implementations of several neural networks architectures;
- to propose the best software/hardware configuration suitable for deployment.

To achieve these goals, an inference tool (specifically tuned for the target system) was developed and multiple others were tested. The developed tool is not only a prototype of a real world inference engine that fits the performance needs of the main project, but was also a test bed for cuDNN and TensorRT, two high performance inference libraries developed by NVidia. Making the inference engine from scratch allows for fine tuning of all involved components, leading to better performance and a fairer testing environment. The key challenges for the project were:

- to test the Jetson embedded system with very limited support documentation;
- software compatibility issues: most available software required to be compiled on the device (with ARM Instruction Set Architecture (ISA));
- to re-implement multiple neural network models on different inference engines.

With these challenges overcome, the project provided an extremely rich testing environment to assess the real world capabilities of the embedded device. It will also shed some light on how different deployment frameworks take advantage of the available on-chip GPU CUDA-cores.

1.2 Document outline

After the current chapter, this document has 7 more; the next one, chapter 2, *The target system*, presents the NVidia Jetson TX2 (the embedded device that will be used in the project), its specifications and its power modes. It also contains the results of an experimental performance evaluation. During this evaluation, the embedded device was tested in multiple power modes and compared against traditional computing hardware to establish a performance baseline.

Chapter 3, *Computer vision with neural networks*, gives an overview of the connection between these two fields and goes in detail on the latter. This chapter justifies the computational demands created by deep learning models, explaining in great detail the operations performed in each layer type.

Chapter 4, *Neural network inference systems*, presents the relevant neural network libraries for the project (cuDNN and TensorRT) and the frameworks that will be used for a comparative evaluation, namely Darknet, Tensorflow and PyTorch.

Chapter 5, *The Deeploy tool*, presents the current state of the developed tool. It starts by explaining it, how it fits in the neural network development cycle and then describes its inner workings by presenting some of its most important classes.

Chapter 6, *Validation and performance analysis*, shows the profiling and fine-tuning steps taken in the development of this tool. At the end of the chapter, a section is dedicated to the validation method employed to attest the correct implementation of the framework.

Chapter 7, *Framework comparison and testing*, starts by explaining the testing environment used, followed by a summary of all the comparisons performed between frameworks.

Chapter 8, *Conclusion*, provides an overview of the process of developing the tool, summarises the most interesting findings and proposes possible tracks for future work on the project.

2

THE TARGET SYSTEM

This project is being developed specifically with vehicles in mind. When building a new system that has to physically fit in such a crowded space as the interior of a vehicle, it is important to take its size, shape and power consumption under consideration. These constrains heavily influence the components selection when developing a new product. Another aspect to consider is that the project is heavily dependent on image processing and object detection which are synonymous with heavy computations. Due to these constrains, the NVidia Jetson TX2 was chosen as being the main computational device coordinating the system.

This device is a System-on-Module (SOM) that incorporates a combination of performance and power efficiency in a small form factor making it ideal for many situations. This system was mainly developed with machine learning, augmented reality and video processing in mind, justifying its vast Input/Output (I/O) and processing power. One of the most important features of the device in this context is its ability to adapt to the project. As seen in figure 1, the device itself does not contain any I/O or cooling system. It is expected for the system integrator to develop and manufacture its own custom board containing only the necessary I/O and the best form factor for the project. Another very important aspect that is not common in embedded devices is the inclusion of an integrated Pascal GPU, further expanding its computational capabilities. Table 1 and figure 2 shed some light into the particular inner working of the device.

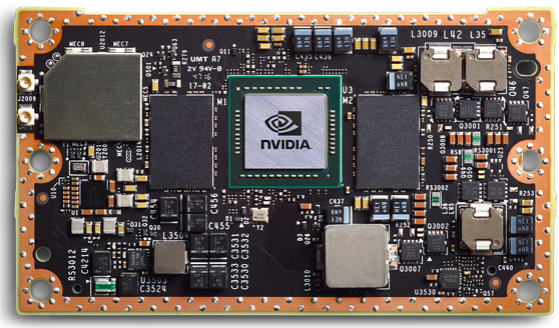


Figure 1: NVidia Jetson TX2

CPU	ARM Cortex-A57 (quad-core) @ 2GHz + NVidia Denver2 (dual-core) @ 2GHz
GPU	256-core Pascal
Memory	8GiB 128-bit LPDDR4 @ 1866Mhz
Storage	32GiB eMMC 5.1
I/O	2x HDMI 2.0 / DP 1.2 / eDP 1.2 / 2x MIPI DSI USB 3.0 + USB 2.0 / UART, SPI, I2C, I2S, GPIOs
Wireless Connectivity	802.11a/b/g/n/ac 2x2 867Mbps / Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet
Power Consumption	7.5W (Typical power consumption)

Table 1: NVidia Jetson TX2 hardware specifications

#	Mode Name	Denver2	ARM Cortex-A57	GPU Freq.
0	Max-N	2.0 GHz	2.0 GHz	1.30 GHz
1	Max-Q	0 GHz	1.2 GHz	0.85 GHz
2	Max-P Core All	1.4 GHz	1.4 GHz	1.12 GHz
3	Max-P ARM	0 GHz	2.0GHz	1.12 GHz
4	Max-P Denver	2.0 GHz	0 GHz	1.12 GHz

Table 2: Jetson TX2 power modes

The most peculiar aspect of the specification sheet is the fact that the board contains two CPU clusters. These CPU clusters have different characteristics and use cases. The Denver 2 cluster has a higher single threaded performance but a lower multi-threaded performance when compared with the Cortex-A57 complex Franklin (2017a). Both of these complexes have vectorial capabilities (NEON, 128-bit wide) which should be utilised for maximum efficiency. These two CPU complexes are completely abstracted away from the programmer. In fact, it is possible to turn on and off any of the CPU complexes without effecting execution of any process, at run time. NVidia also developed custom silicon to maintain cache coherency between the CPU complexes (CPU switch fabric in figure 2).

The next interesting specification is the GPU. It has all the capabilities of a conventional desktop GPU like extensive compute Application programming interfaces (APIs) and libraries like CUDA. The graphical memory is shared with the main system memory (as seen in figure 2) which may improve transfer times between GPU and CPU via NVidia's *Unified Memory* technology. It is also important to mention the vast I/O supported by the device (as depicted in table 1) and its power consumption of just 7.5W on a typical load.

Finally, the target system also comes equipped with 5 different power modes (table 2), to suite possible power consumption or heat dissipation constraints. At run time, by executing the command `sudo nvpmodel -m <# desired mode>` the power mode can be changed, further adapting the board to the needs of the system. The power mode essentially works by throttling or even disabling certain components of the board (as summarised in table 2).

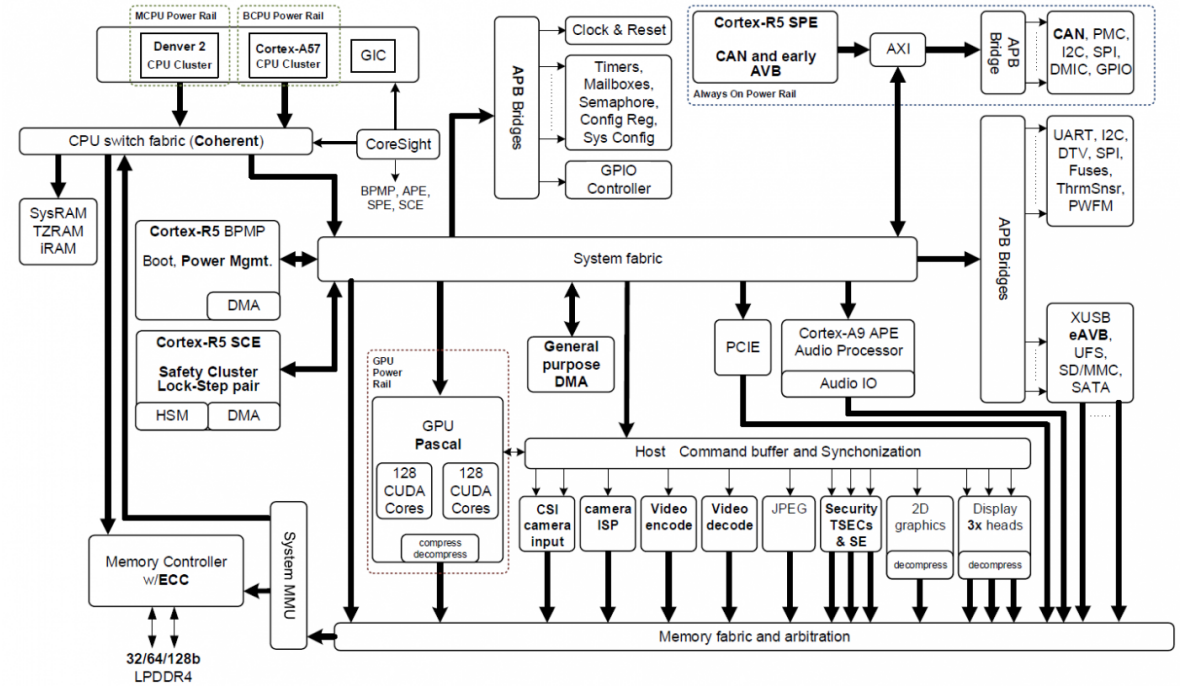


Figure 2: NVidia Jetson TX2 block diagram (from Jetson TX2 specification)

2.1 KEY PERFORMANCE CHARACTERISTICS

To create a more testable and systemic working environment, it is important to focus on key features. For a performance critical deployment, the hardware characteristics worthy of highlight are:

- **Shared Memory between CPU and GPU**

  As seen in figure 2, the CPU and the Pascal GPU share memory and access it via the *memory controller*. In a traditional High Performance Computing (HPC) device, the CPU and the GPU have each their dedicated memory, forcing any communication between them to occur via the Peripheral Component Interconnect Express (PCI-e) bus.

  Here, both computing devices share the same memory, possibly rendering some memory management operations redundant. This idea becomes even more apparently feasible due to the Unified Memory technology implemented by NVidia Harris (2017). This allows the developer to neglect memory management between the CPU and the GPU, making any memory allocation to be accessible, seamlessly, by both devices.

  On a conventional system, this technology is not expected to create any performance enhancements since data migrations between CPU and GPU, although automatically

Chip	FP16/FP32	FP64/FP32
GP100 (Tesla P100)	2:1	1:2
GP10B (Jetson TX2)	2:1	1:32
GP102 (GTX 1080ti)	1:64	1:32

Table 3: Pascal floating point relative performance

managed, are still happening. But on a device with shared memory, depending on how this technology is implemented, it can eliminate unnecessary data movements.

- **Half-precision floating point computation**

  With the introduction of the Pascal architecture came full support for half-precision floating point operations (fully compliant with IEEE 754 Zuras et al. (2008)), but with some performance related caveats. Maxwell, the preceding GPU architecture, also supported Half-precision floating-point (FP16) operations but only on one chip: GM20B, the chip that equipped Jetson TX1, the Jetson TX2 ancestor Ho and Wong (2017).

  When it comes to performance, the Pascal implementation of FP16 is heavily dependent on the chip in question. GP100, the fastest iteration of Pascal and the chip that equips the Tesla P100 GPU, has an exceptional FP16 support, with twice the theoretical throughput when compared to single precision operations. This is achieved by making the CUDA cores capable of concurrently process two FP16 operations Harris (2016). This technique is also applied in GP10B, the chip that equips Jetson TX2 Larabel (2017); Franklin (2017b). On the other hand, GP102, the chip used in the high end Pascal consumer Geforce GPU does not handle half precision as efficiently. In fact, using half precision yields less performance since only one CUDA core per SM is capable of processing two FP16 operations concurrently Smith (2016).

  Table 3 summarises these implementation caveats. Even though all these chips are Pascal, they have extremely different behaviour depending on the floating point precision, highlighting the importance of knowing the hardware for performance critical implementations. The theory behind lower precision floating point operations on neural networks is explained in more detail in latter sections.

- **Power modes**

  It is also very important to determine how the power modes will impact performance, especially the ones that throttle the GPU frequency. These are particularly important if the project gets constraints when it comes to power consumption or heat dissipation.

2.2 EXPERIMENTAL PERFORMANCE EVALUATION

A practical performance evaluation is important to determine where the device is situated when compared with common hardware. This helps to level expectations and validates the actual achievable peak performance.

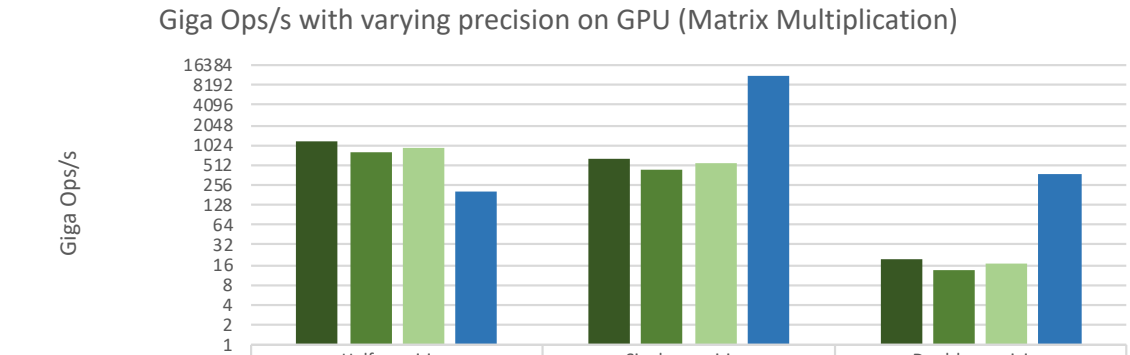
Matrix multiplication is the standard operation used to determine peak achievable performance. Tensorflow was initially considered to execute this test since this framework is capable of performing these operations at varying floating precisions and is also capable of using NVidia GPUs. However, this frameworks, probably to overcome poor half precision support of consumer cards, performs these computations at single precision, even when specifically programmed not to.

For this reason, a simple C++ code snippet was developed to accomplish the desired operation: measure peak floating point operations per second when performing matrix multiplication at varying floating point precisions. The code is based in *cuBLAS*, a library that implements the standard basic linear algebra subroutines developed by NVidia for their GPUs Nvidia (2008).

Figure 3 shows the measured throughput of Jetson TX2 and a GTX 1080ti while operating on two square matrices with 4096^2 elements each. The results were taken after a warmup multiplication call and are the average of 10 executions.

Since both these devices can execute, on single precision floating points, a Fused Multiply–Add (FMA) instruction Whitehead and Fit-Florea (2011) per cycle per CUDA core, their theoretical peak throughput can be calculated by $(\#CUDA\_cores) * Frequency * 2$. This puts the Jetson TX2 (Max-N) at a theoretically peak throughput of 665.6 GFLOPS ($256 * 1.3 * 2$) and the GTX 1080ti at 11339 GFLOPS ($3584 * 1.582 * 2$). Multiplying by the factors in table 3 calculates the throughput with other floating precisions. These theoretical results were corroborated by the measured performance summarised in figure 3, meaning that the theoretical peak throughput is achievable in practice during matrix multiplication operations, ruling out any possible memory bottleneck. This is important since the most common and computationally demanding layer types (convolutional and fully-connected) can be implemented via these matrix operations.

Only the power modes that interfere with the GPU frequency were tested, and the results, as expected, scaled linearly with its change. The most important takeaway from the measurements is the fact that the embedded device beats the top of the line Pascal consumer GPU on half-precision floating point operations (1180 GFLOPS *vs* 198,44 GFLOPS). This emphasises the importance of favouring these types of operations in this device and avoiding them in the dedicated GPU. This is important to note since neural network training is normally performed in the later. This way, to fully take advantage of both architectures, training should be performed in single precision and the resulting weights should be con-



	Half-precision	Single-precision	Double-precision
Jetson TX2 (Max-N)	1180.30	619.53	19.68
Jetson TX2 (Max-Q)	804.68	419.26	12.93
Jetson TX2 (Max-P Core All)	946.55	547.85	16.95
GTX 1080ti	198.55	11406.10	370.05

Figure 3: NVidia Jetson TX2 GPU floating point throughput with varying precision and power modes (compared with GTX 1080ti)

verted to half precision for final deployment. This may result in problem at inference time due to the change in precision between training and inference. This topic will be studied with more detail along the work.

The same series of tests were also performed on the CPU side. To this effect, a small C++ application was also developed, this time based on *OpenBLAS*, an optimised Basic Linear Algebra Subprograms (BLAS) library Xianyi et al. (2014). Here, all the available power modes were relevant, since all of them generate unique combinations of core frequencies.

The results, shown in figure 4, reveal a very uninteresting Denver core cluster when compared with its ARM counterpart. Even accounting for the core count difference (4 *vs* 2) the Denver cluster turned out to be extremely inefficient (44,28 GFLOPS *vs* 4,62 GFLOPS in single-precision), to the point where it seems more suitable to have it turned off. As for the general performance when compared with a high performance consumer CPU (with the same core count), the embedded device revealed, as expected, much inferior (up to 9x slower in MAX-N mode). This highlights the importance to explore the GPU for the neural networks inference tasks, leaving the CPU for other less computationally demanding workloads.

For reference, table 4 displays the specifications of the workstation used in the comparison. This system will also be used for testing in later chapters and is the system used, currently, to perform model development/training at Bosch.

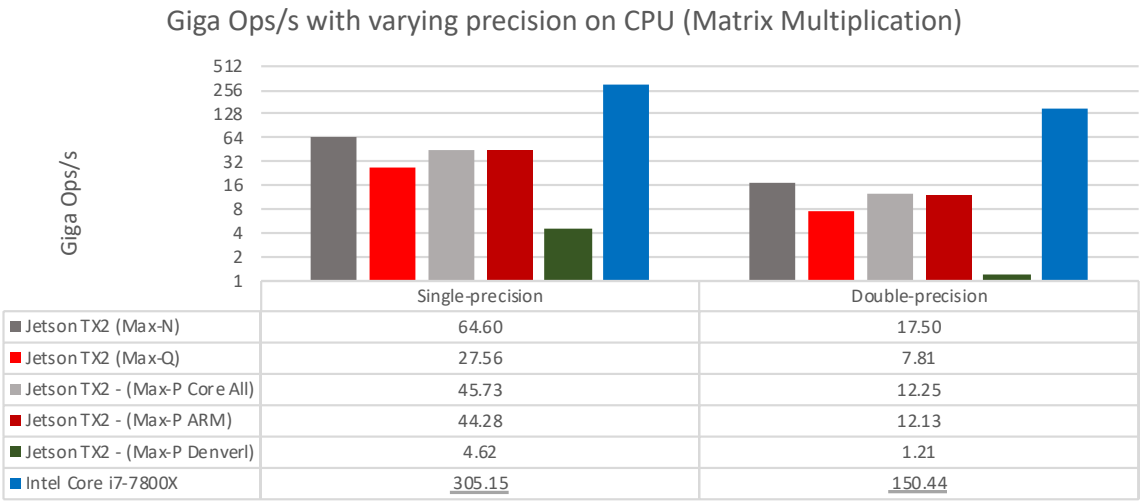


Figure 4: NVidia Jetson TX2 CPU floating point throughput with varying precision and power modes (compared with an Intel Core i7-7800X CPU)

CPU	
Model	Intel® Core™ i7-7800X
Architecture	Skylake
Cores	6
Cache L1	12x 32KiB (6x inst. + 6x data)
Cache L2	6x 1MiB
Cache L3	8,25MiB Shared
Frequency	3,5 Ghz
SMT	Yes (2 way)
ISA Extensions	SSE4.1/4.2, AVX2, AVX-512
RAM	4x 8GiB DDR4 @ 2133 MHz

GPU	
Model	GTX 1080ti
Architecture	Pascal
#CUDA Cores	3584
Frequency	1,480 Ghz
Memory	11GiB GDDR5X

Table 4: Workstation system specifications

3

COMPUTER VISION WITH NEURAL NETWORKS

Computer Vision is the computer science field that is responsible for the transformation of data from video or still images into a decision or a new representation Kaehler and Bradski (2017). This may seem like a trivial task. Humans can very easily interpret visual data and extract high level knowledge from it. For a computer however, this is not as straightforward since it must be able to interpret a set of bytes that represent, pixel by pixel, the image as a whole. [1] For this reason, until now, there is no definitive solution for problems like object detection.

Neural network based approaches have proven to perform better than traditional computer vision techniques (e.g.: SIFT) in object detection tasks Krizhevsky et al. (2012); Simonyan et al. (2013). In fact, neural networks have proven to be fundamental for a variety of different tasks that will be useful for the project (e.g.: Pose estimation Cao et al. (2016); Wei et al. (2016), Object Detection Redmon et al. (2016); Redmon and Farhadi (2018); Howard et al. (2017); Krizhevsky et al. (2012) and Natural language processing Gillick et al. (2015); Józefowicz et al. (2016)). For these reasons, Deep Neural Networks deployment on Jetson TX2, the main focus of the dissertation, is critical for the main project.

3.1 DEEP NEURAL NETWORKS

There is no Machine Learning related topic more popular than Deep Neural Network (DNN). This research field has inspired plentiful publications and witnessed numerous advances over the last decade Schmidhuber (2015). Like most ML approaches, Deep Neural Network are capable of learning a new task without being explicitly programmed to.

In particular, this dissertation is going to focus on neural networks that have to be trained with big amounts of labelled data before they can be used for inference (supervised learning Mohri et al. (2012)). This step can not be ignored when deploying a neural network but it is not going to be addressed here since the training will be performed in dedicated

1 This is a simplified view of the matter. There are factors like image format, resolution and number of channels that come into play.

workstations. The trained model will then be transferred to the embedded device, ready to infer on new data.

Though not a recent topic by any means (with some literature dating back to the 40's), Deep Neural Networks have increasingly become a very hot research field. This increase in popularity can be attributed to the increase in availability of both software and hardware that facilitate the development, testing and training of DNNs.

On the software side, frameworks like Tensorflow Abadi et al. (2016), *Caffe* Jia et al. (2014) and *MXNet* Chen et al. (2015) coupled with accessible and forgiving programming languages like *Python* or *R* lower the barrier of entry. And if software aids with the conception and development of new architectures, recent hardware (particularly GPUs) lower the training time allowing for more architectures to be tested or further tuning of existing ones Markidis et al. (2018); Wu et al. (2016); Deng et al. (2014).

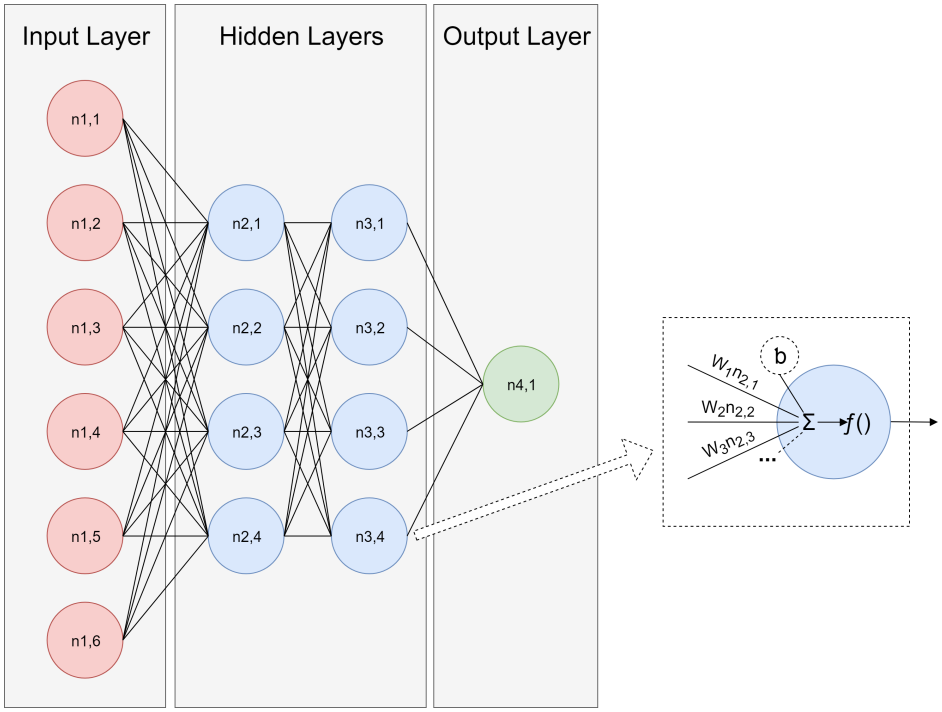


Figure 5: Example of a Fully Connected Neural Network

Artificial Neural Networks (ANNs) were initially developed to model biological neural systems. They can be described as biologically inspired computing models that are composed of collections of interconnected neurons Li and Karpathy (2015). Figure 5 shows a very basic fully connect deep neural network.

Like illustrated, a neural network is typically organised in layers Li and Karpathy (2015). Here, only fully-connected layers are used, hence the name. The deep terminology is not applied here since the network is relatively shallow, containing only two hidden layers. In the next chapter, other layer types that typically compose a DNN will be explained. Another

aspect clearly depicted in the figure is the concept of a neuron. A neuron can have multiple inputs and a single output (that can serve as input to one or more neurons). When training, all the neurons that compose the network ”learn” or readjust their weights and bias in an attempt to get closer to the correct known output for a given input. When making an inference, a neuron sums all of its inputs (multiplying each by its respective, previously learnt, weight), adds its (also learnt) bias and feeds this result to its predefined activation function.

3.1.1 *Neural network layers*

This section presents the most common types of layers that typically form a Neural Network. But, to fully understand layers, their different types and what they do precisely, it is important to understand the concept of a tensor as presented in Abadi et al. (2016). Essentially, a tensor is a typed multidimensional array that represents the input and output of a layer. Figure 6 illustrates how a tensor with dimensions 3x5x5 can fully represent the image depicted (with a resolution of 5x5). Keeping the analogy between images and tensors, their dimensions can sometimes be named Height, Width and Channel accordingly.

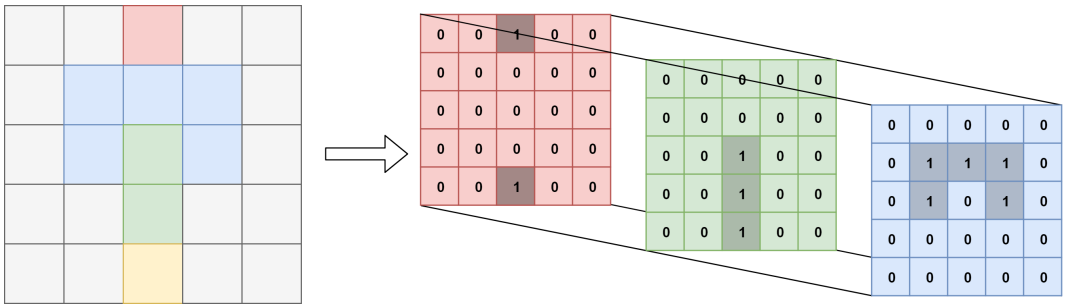


Figure 6: Tensor example

The next section will briefly present some of the most common layer types.

Activation layer

The activation layer if one of the most basic types of layer. It simply applies a predefined function to all elements of the tensor, maintaining its shape and type. They are rarely seen in the specification of a neural network since they are normally applied implicitly after a fully connected or a convolution layer, becoming part of the specification of these layers.

The most important part of the an activation layer is the activation function. It is important for this function to be non-linear since otherwise the computation performed by interconnected neurons could be reduced to a simple linear algebra operation Sze et al. (2017).

Here are some of the most common activation functions:

1. **Sigmoid**
   Sigmoid was one the most common activation function used in the beginning of ANNs. It mimics the firing rate of a biologic neuron by firing (1) or not firing (0). It essentially reduces any input to a value between 0 and 1. Its mathematical formula is $\sigma(x) = \frac{1}{(1+e^{-x})}$ Li and Karpathy (2015).

2. **Tanh**
   The formula for *tanh* is very close to the formula of *sigmoid*: $tanh(x) = \frac{2}{1+e^{-2x}} - 1$. Here, the input gets reduced to a value between -1 and 1. Since it is zero centered, it is always preferred over *sigmoid* since this facilitates backpropagation during training Li and Karpathy (2015).

3. **ReLU**
   The Rectified Linear Unit has become one of the most used activation functions. It simply is $f(x) = max(0, x)$ which effectively thresholds any input to a minimum of 0. Its main advantages over *Sigmoid* and *Tanh* are greatly simplified computation (no exponential or fractions) and a faster converging rate when training Krizhevsky et al. (2012).

4. **Leaky ReLU**
   Though inconsistently, many have reported success by replacing ReLU with Leaky ReLU. Here, negative inputs are multiplied by a (predefined or trainable) α: $f(x) = max(\alpha x, x)$ Li and Karpathy (2015).

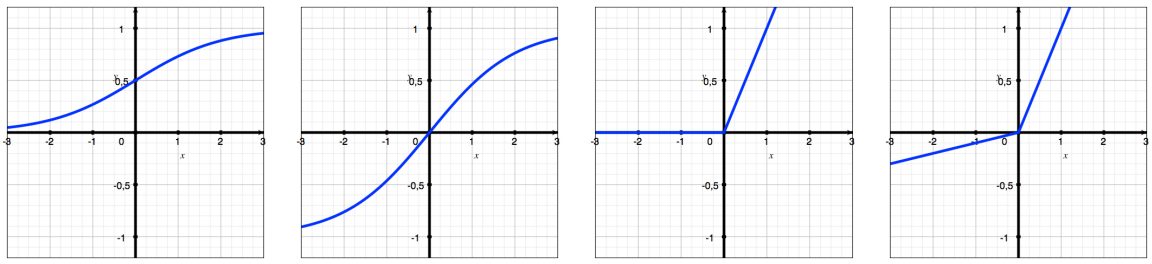


Figure 7: Common activation functions; from left: Sigmoid, Tanh, ReLU, Leaky ReLU

Fully connected layer

A fully connected layer is a layer where each of its neurons is fully connected to all the outputs of the preceding layer. Currently, this layer is typically used as the final layer on convolutional networks (Howard et al. (2017); Simonyan and Zisserman (2014)). The goal is for these layers to learn meaningful information from the high level features extracted by the preceding layers.

Another important aspect of this layer type is the very high number of trainable weights. This makes it a very computationally heavy layer type in terms of number of operations and in number of weights to be stored. Fortunately, this operation can be implemented as a matrix multiplication, a highly studied and optimised algorithm in almost any computer system Sze et al. (2017). Looking at a practical example, VGG-16's first fully-connected layer is connected to a layer that outputs a 512 x 7 x 7 tensor and converts it to a 4096 x 1 x 1 one. This means that it has $(512 * 7 * 7) * 4096 + 4096 = 102,764,544$ trainable weights Simonyan and Zisserman (2014).

It is apparent that this layer is not ideal for visual recognition tasks since the input tensors are expected to be big (depending on resolution of the input image). Besides the weight problem, fully-connected layers do not take any advantage of the spatial properties of an image. For instance, if the goal is to develop an object detector, the features that help to detect it are relevant in every part of the image, thus, many redundant weights will be unnecessarily learnt. Next section presents a layer type more suited for theses tasks.

Convolutional layer

Convolution layers are the fundamental building blocks of convolutional neural networks. These layers have proven to be extremely competent in object classification and localisation tasks when compared to more traditional techniques. Like described by LeCun et al. (1989), their goal is to extract local features and to combine them into higher level features. They also assumed that the object can appear in any location of the input image and that this location should not affect the final object classification.

These goals are built into convolutional networks by chaining multiple convolutions. Each convolution is responsible for extracting features from its inputs, hence, the concatenation of multiple layers like these results in the desired hierarchical and location agnostic feature extraction system.

The inner workings of a convolutional layer are relatively simple. The convolution type presented in this dissertation is the one used throughout all the networks implemented but other types of convolution exist. A convolution can be thought of as a set of filters. Each filter will interact with every channel with different weights and generate a new channel in the output tensor. Every filter will convolve each input channel with its respective weights by multiplying, scalar by scalar, the weights and a subsection of the input channel. The multiplied number are later summed and a bias is finally added to generate the final result. This process is repeated for every filter Li and Karpathy (2015). This is depicted in figure 8.

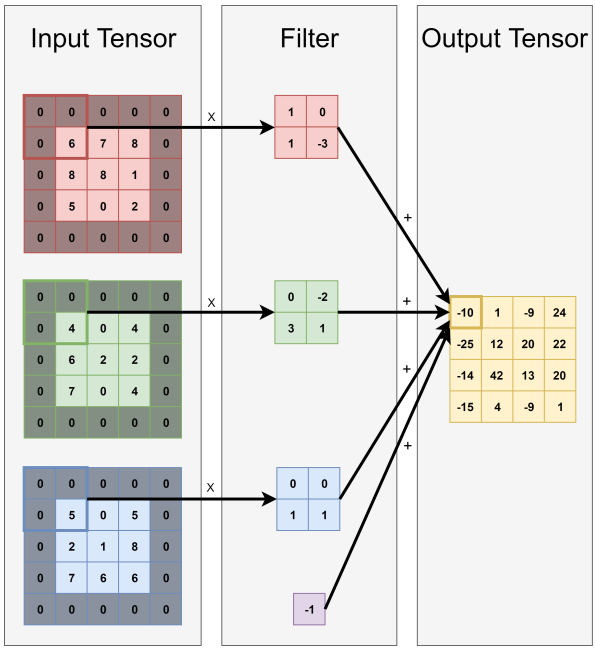


Figure 8: Convolutional operation

Finally, it is important to understand all the specifications required to define a convolutional layer. The convolution in figure 8 defines a convolution that only has a filter (K), is applied with a stride (S) of one, has a size (F) of 2 and a padding (P) of 1. Considering the input tensor dimensions Width, Height and Channels as W_i, H_i, C_i respectively, the output tensor dimensions (W_o, H_o and C_i) will depend on the convolution specification the following way:

- $W_o = (W_i - F + 2P)/S + 1$
- $H_o = (H_i - F + 2P)/S + 1$
- $C_o = K$

Pooling layer

The purpose of a pooling layer is to reduce the spacial size of the representation without loosing too much important information. A very common type of pooling layer is a Max Pooling Layer. This type of layer operates independently between each slice of the input volume, dividing then in subregions and choosing the biggest value in each region for the output volume. This reduces the width and the height of the input but it does not change its depth. Figure 9 illustrates a 2 by 2 max pooling layer applied to one slice of the input volume.

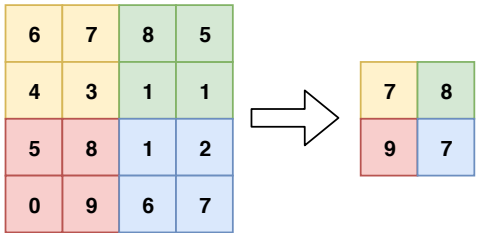


Figure 9: Pooling layer with (stride 2 and size 2)

Batch normalisation

As training progresses, the weights of each layer change which leads to a distribution shift of their outputs. A change in a single weight is amplified by all the subsequent layers, forcing them to continuously adapt to a new distribution during training Ioffe and Szegedy (2015). This forces the use of lower learning rates to minimise this effect, increasing training time.

Batch normalisation alleviates this problem by fixing the distribution of the tensors to a zero mean and unit variance. This is just the first step of the normalisation layer. It also introduces a pair of trainable parameters that scale and shift the normalised values. If these parameters were not introduced, the *sigmoid* activation function, for example, would be reduced to the linear part of its activation range. Note that these parameters can, in theory, revert the first normalisation step performed. In practice, this does not happen very often. Typically, this layer is used just before the activation function.

This layer type is normally applied to the 3D tensors outputted by convolutions. These tensors are composed by a channel dimension (C), width (W) and height (H). Since convolution layers apply independent filters for each channel, batch normalisation is normally applied along this dimension. To apply batch normalisation, during training, the mean (μ), variance (σ^2), scale (γ) and shift (β) have to be determined. Since, in this example, batch normalisation will be applied along the channel axis, all of these variables have to be vector of length C. To calculate the output tensor Y, two steps have to be performed: the normalisation step followed by the affine transform step. Considering X and T as input and intermediate tensors respectively, the batch normalisation is performed as followed:

$$T_{cwh} = \frac{X_{cwh} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} \text{ for } c = 1, ..., C, w = 1, ..., W, h = 1, ..., H$$

$$Y_{cwh} = \gamma_c T_{cwh} + \beta_c \text{ for } c = 1, ..., C, w = 1, ..., W, h = 1, ..., H$$

Figure 10: Batch normalisation formula

with ϵ being a very small value to avoid division by 0.

Softmax

When trying to develop a network to classify if a certain image contains a person, it is intuitive to think of the output as a probability. For instance, if the network outputs 0.75, there is a 75% change of the object being present a 25% change of it not being present. Softmax applies this concept to multi-class classification systems. It assigns probabilities to the raw outputs of the network, making the sum of all the outputs equal 1 Bishop (2006).

Given an input vector V with n elements, the resulting n elements of the output vector R are calculated as:

$$R_i = \frac{e^{V_i}}{\sum_{k=1}^{n} e^{V_k}} \text{ for i = 1, ..., n}$$

3.1.2 *Batch size*

Batch size dictates the number of input tensors the network processes at a time. It has a very different meaning depending on the direction of the propagation: backwards (training phase) or forwards (inference phase).

During the training phase, batch size specifies the number of input elements (e.g.: images) that contribute to every weight adjustment during back-propagation. A big batch size normally equates to a more efficient training since each weight update takes a bigger number of input elements into consideration, leading to more precise adjustments. The disadvantage of big batch sizes is that memory consumption raises accordingly.

At inference time, the focus of this work, batch size is less important. Analogous to the training phase, here this setting dictates the number of elements that are processed per forward pass. This does not have any impact in the results generated by the network but can have some effect in the time it takes to perform inference. Typically, making a single inference with a big batch size is more efficient than making multiple inferences with small batch sizes. This can be explained since a big batch sizes give more alternatives for the engine to explore memory optimisations like selecting cache friendly algorithms or latency masking. This work will test this effect in the embedded device by making measuring inference times at different batch sizes.

3.1.3 *Lower precision floating point arithmetic*

Half-precision floating point is a binary floating point format that is able to represent floating-point values in just 2 bytes. This format is part of the IEEE-754 Zuras et al. (2008) standard and is named `Binary16`. This smaller memory footprint comes, at the cost of precision when compared to the more usual single-precision representation.

To fully understand how a lower size influences precision, it is important to understand how floating point representation works. The following formula, along with figure 11, depict how normalised values are calculated:

$$V = (-1)^{sign} * (1.mantissa) * 2^{exponent-127}$$

This formula indicates that it is the mantissa that dictates the number of significant digits that the format is able to represent. With a mantissa with 10 bits (plus the hidden bit), FP16 is able to store numbers with 3 significant digits ($log_{10}(2^{11}) = 3,31$), while Single-precision floating-point (FP32) can represent between 7 and 8 significant digits ($log_{10}(2^{24}) = 7,22$). Table 5 contains the absolute precision of both floating point representations are various ranges, giving a more clear understanding of the subject.

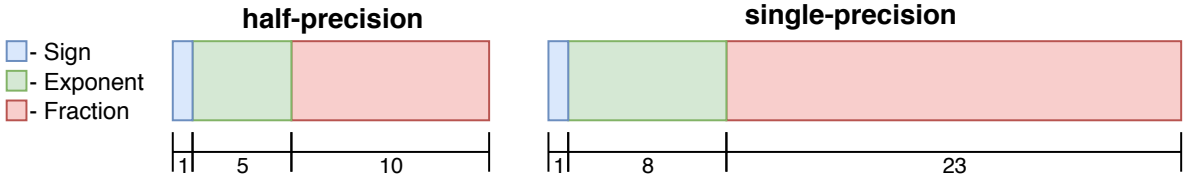


Figure 11: Half-precision *vs* single-precision size and bit allocation

Exponent	Range	Half-Precision	Single-Precision
0	[1, 2)	0,0009765625	0,0000001192092
1	[2, 4)	0,001953125	0,0000002384185
2	[4, 8)	0,00390625	0,0000004768371
3	[8, 16)	0,0078125	0,0000009536743
4	[16, 32)	0,015625	0,000001907348
8	[256, 512)	0,25	0,00003051757
15	[32768, 65536)	32	0,00390625
16	[65536, 131072)	—	0,0078125

Table 5: Smallest detail expressible by half-precision and single-precision at varying ranges

This representation is not only highly efficient in the embedded device in question (as previously stated), but it is also widely used in the neural network field. Due to their computationally expensive nature, neural networks are typical the target of very aggressive optimisations. One of the most common one is to perform the training and inference stages in half-precision. This typically comes at the cost of slightly lower inference quality but a much higher efficiency (only if the hardware fully support this representation).

Even though these formats have very different precisions, half-precision has revealed to be extremely adequate for neural networks Gupta et al. (2015). The values that need to be represented inside a neural network are typically small numbers, between -1 and

1. This is due to the activation functions that are normally used and to layers like batch normalisation that limit the absolute size of the tensor values. It is in this range, similarly to single-precision, that half-precision has its highest absolute precision, making it ideal for this use case.

This work will also explore the viability of performing training and inference at different precision. This is relevant for situation where the hardware that is used for training does not benefit from the lower precision floating point arithmetic, while the deployment device does (or vice versa).

3.2 Deep neural network models for computer vision tasks

The most important networks for the project are the ones related to object detection/classification with parts of VGG-16 even being used as the basis of OpenPose (a pose estimation network Cao et al. (2016)). The following sections presents in more depth this topic and the most relevant network architectures.

3.2.1 *Object classification*

Object classification differs from object detection in that it only classifies the main object in the centre of the input image Everingham et al. (2010). This way, the output of the networks can be a simple one dimensional tensor with a length equal to the number of classes to classify.

Typically, these networks are composed of a set of convolution layers (for feature extraction) connected to a final fully connected layer to make the actual object classification (like in LeCun et al. (1998)). This meta architecture will be more clear after presenting VGG-16 and ResNet-50 in the next sections.

VGG-16

VGG-16 implements a very typical neural network architecture for object classification. As stated before, this network can be though as having two parts: feature extractor module and a classifier module Simonyan and Zisserman (2014); LeCun et al. (1998). In this case, feature extraction is performed by a combination of convolution layers and max pooling layers just like depicted in figure 12. After feature extraction, classification itself is executed by two fully-connected layers followed by a softmax layer to produce the final class probability distribution.

Although very influential for the time, this network is not particularly useful for the project since it can only perform image classification. However, its efficient implementation

is very important since its first 10 layers are utilised by *OpenPose* for feature extraction purposes Cao et al. (2016)

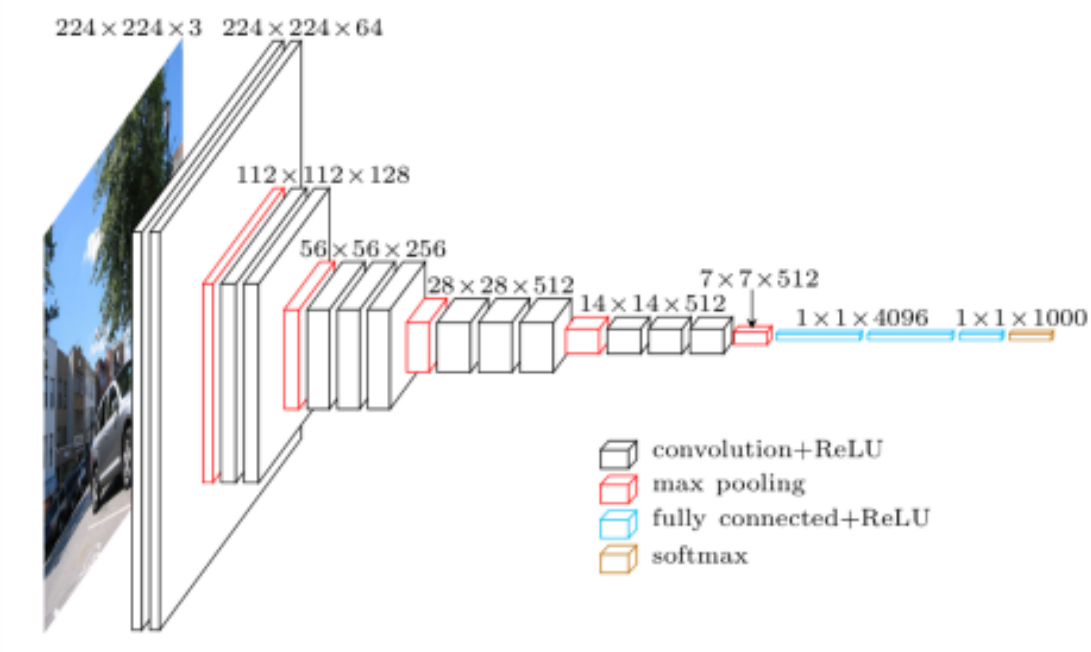


Figure 12: VGG-16 architecture (from leonardblier (2016))

ResNet-50

ResNet is the abbreviation of Residual Network, where some output layers skip the subsequent layer, connecting to some other, deeper, layer (shown in figure 13).

As networks get deeper, training becomes more difficult due to the accumulation of multiplications leading to values close to zero and the tendency for the deeper models to overfeat. These residual connections create shorter paths withing the network, eliminating both of these issues.

The variant in question, ResNet-50, employs the same basic principals that VGG-16 implements, with a sequence of convolution layers followed by a fully connected layer at the end. Being ResNet, obviously, has the difference of using residual connections. This network model was chosen due to its popularity and availability in the frameworks tested.

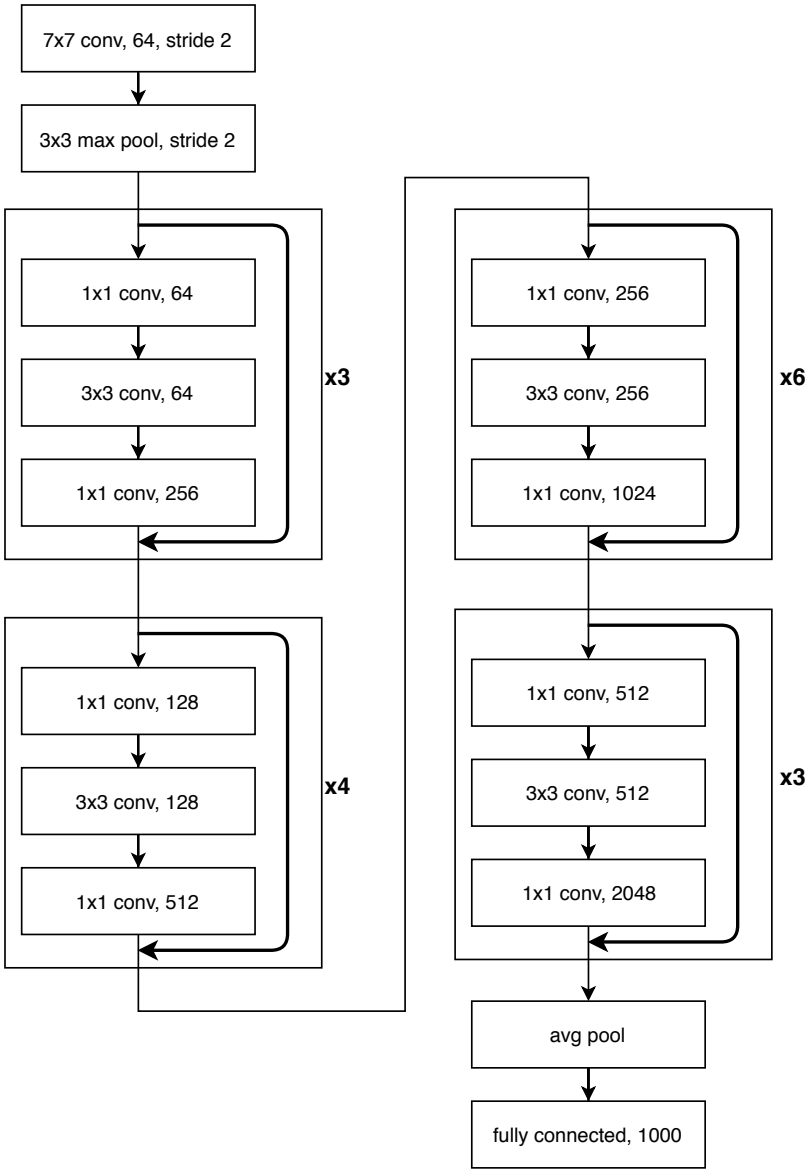


Figure 13: Simplified ResNet-50 architecture overview

3.2.2 *Object detection*

Object detection enables some functionalities, like alerting for left behind objects by the occupants or for garbage left inside the vehicle.

Object detection goes beyond object classification since it can detect, identify and locate multiple objects in an input image Girshick et al. (2014b). Although still an open computer vision problem, working solutions have been proposed and are presented in the next section.

R-CNN

As shown in figure 14 this first approach to object detection tries to solve this problem in two stages: a region proposal stage and an image classification one Girshick et al. (2014a). This method essentially reuses the image classification techniques to perform object detection.

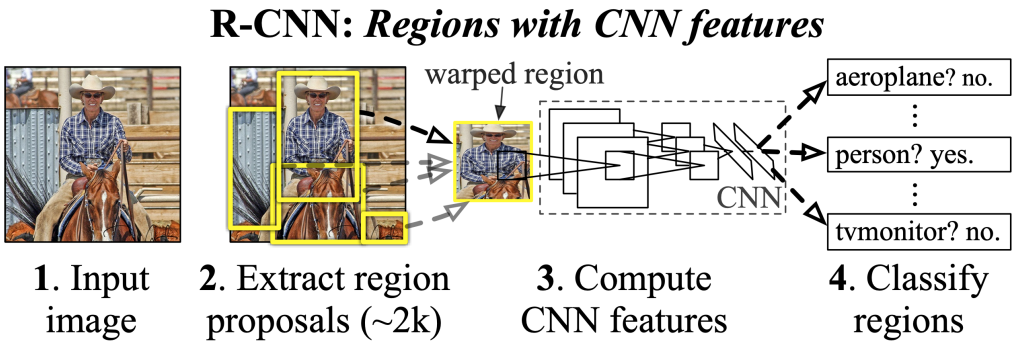


Figure 14: R-CNN overview (from Girshick et al. (2014a))

The first stage (region proposal) applies selective search Uijlings et al. (2013) to generate approximately 2000 regions per image. These regions may not have a shape compatible with the classification network and, for this reason, need to be warped to be classified. This process is explained in figure 14.

This method has obvious disadvantages like the high computational cost and the fact that the classifier and the region proposal have to be trained separately.

Fast R-CNN

Fast R-CNN improves on the previous method by applying the region proposal algorithm to the result of the first convolution layers Girshick (2015). This way, the classification stage does not need to repeatedly execute the first convolution layers, greatly improving performance. This method is depicted in figure 15.

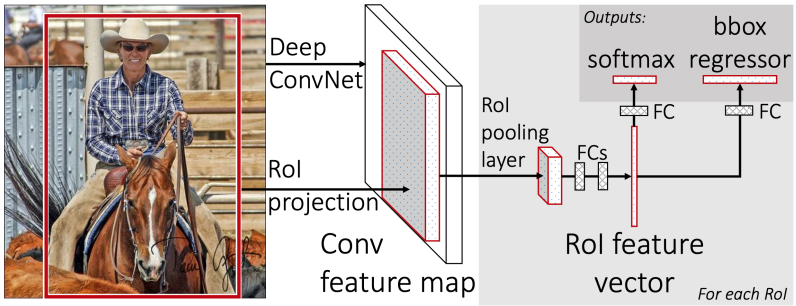


Figure 15: Fast R-CNN overview (from Girshick (2015))

Faster R-CNN

Faster R-CNN goes a step further and abandons the region proposal algorithms, leaving this task to a neural network. This network generates regions and gives them an objectiveness score, essentially predicting the probability of the regions containing an object. Then, the regions with the highest probability of containing an object, are fed to a classifier network.

This technique, depicted in figure 16, largely improves performance over the previous methodologies.

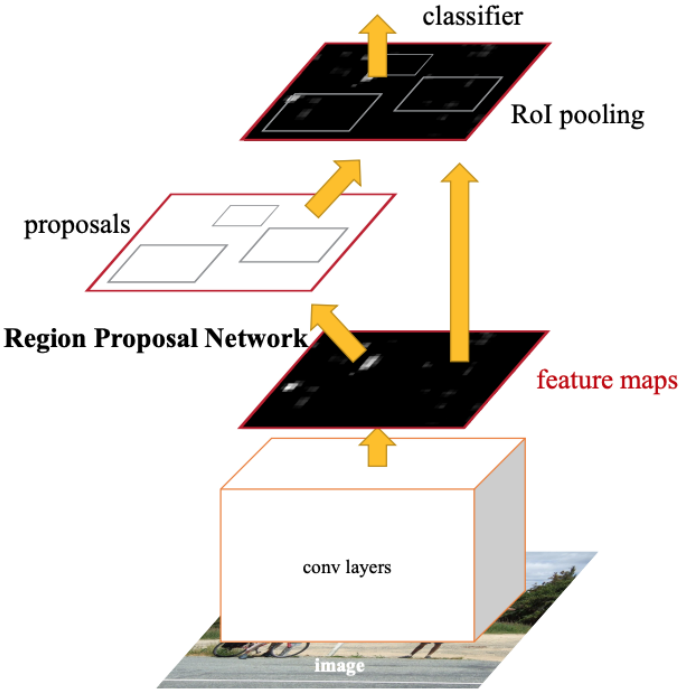


Figure 16: Faster R-CNN overview (from Ren et al. (2015))

YoloV3

Due to its importance for the main project, this network architecture was, initially, the only focus of the work in this dissertation. YoloV3 is a very lightweight neural network for object detection, that distinguishes from the previous architectures due to only being composed of a single network Redmon and Farhadi (2018). As the full name implies (You Only Look Once), this network is able to perform detection on a given image without having to resort to classify multiple areas. In fact, this network does not even contain fully connected layers, typical of classification networks.

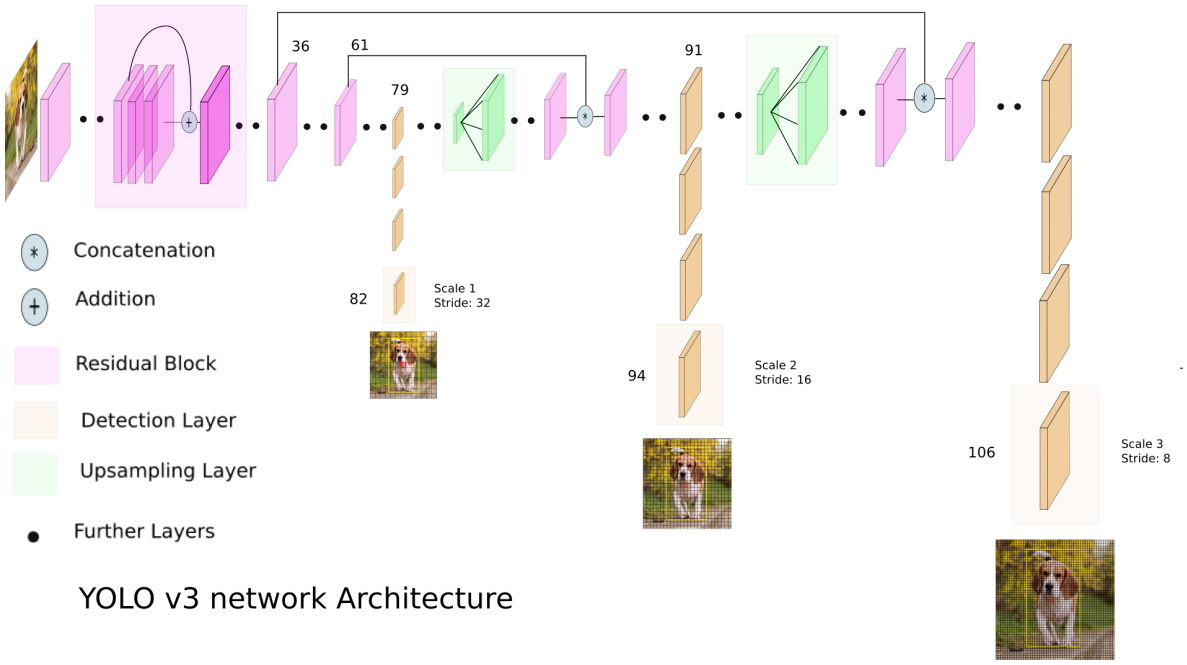


Figure 17: YoloV3 overview (from Kathuria (2018))

At a high level, this network can be decomposed in two parts: the feature extractor and the detection kernels. The feature extractor used in YoloV3 is the one found on Darknet-53, a classification network designed by the same author. This is considered the backed-end of the YoloV3 architecture but can be interchanged in other feature extractors. Its job is to ease the subsequent classification (or detection) stage by detecting and highlighting features in the image that help to distinguish between objects.

Similarly to VGG-16, this feature extractor is mainly composed of convolution layers and accepts input images of size 416 by 416. The training process of this feature extractor can be performed independently from the rest of the network, in which case, it is trained as a classification network, with fully connected layers in the final stages. The feature extractor creates a feature map of 13 by 13 by 255. This is the input to the final detection stage.

At the detection stage, every section of the 13 by 13 feature map gets independently classified and scored. This can be seen as the initial input image being divided into a 13 by 13 grid and independently classified. This stage leverages the way convolution layer operate to perform this classification task across all of the "cells". Lastly, as depicted in figure 17, this network performs this detection step at three different scales, improving detection rates of objects with different sizes.

3.2.3 *Computer vision metrics*

Like any scientific work, computer vision also relies on metrics to measure and compare the performance of its solutions. Here these metrics are particularly important since they will ensure the correct implementation of the models.

In an object classification scenario, the network usually attributes a label to an image by assigning probabilities to each of its known labels of being the correct one. So, by sorting the output in descending order, it is possible to determine what the network thinks are the most correct guesses. *Top-5* and *top-1* error rates, just like the name implies, assesses the fraction of test images that did not see their labels among the top 5 or the top 1 guesses Krizhevsky et al. (2012), respectively. For reference, table 6 contains some top-1 and top-5 error rates for common classification networks on the *ImageNet* Deng et al. (2009) dataset.

Network	Year	Top-1 Error (%)	Top-5 Error (%)
ResNet - 50	2015	24,1	7,1
VGG - 16	2014	28,5	9,9
VGG - 19	2014	27,3	7,1

Table 6: Top-1 and top-5 Error rates on ImageNet validation dataset. (Adapted from `https://keras.io/applications/`.)

When the subject is object detection, it is important to, not only determine if the model correctly identifies the objects, but it is also important to determine if it is capable of locating them correctly as well. This is usually accomplished via a metric called mAP. To understand mAP it is important to understand the concept of Intercept over Union (IoU). IoU is used when there is a need to calculate by how much two regions overlap and, like the name implies, it can be calculated by dividing the interception area by the union area of these regions. This is illustrated in figure 18: mAP is the average of mAP_{50}, mAP_{55}, mAP_{60}, ..., mAP_{95}, with mAP_X being the percentage of predictions that have an IoU larger than or equal to X with the ground truth Lin et al. (2014a).

Figure 18: Intersect Over Union (IoU)

For reference, table 7, taken from Redmon and Farhadi (2018), contains the mAP values for common network architectures on COCO dataset Lin et al. (2014b).

Network	AP (%)	AP_{50} (%)	AP_{75} (%)
YOLOv2 Redmon and Farhadi (2017)	21,6	44,0	19,2
YOLOv3 Redmon and Farhadi (2018)	33,0	57,9	34,4
SSD513 Liu et al. (2016)	31,2	50,4	33,3
RetinaNet Lin et al. (2018)	40,8	61,1	44,1

Table 7: mAP results for common object detection networks on COCO dataset

For object detection purposes, it is also important to take into consideration the amount of wrong detections performed by the network (called "false positives"). For that reason F1-score will also be used to measure the correctness of the implementation.

F1-score is calculated as follows:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

with *recall* and *precision* being calculated as:

$$precision = \frac{truepositives}{truepositives + falsepositives}$$

$$recall = \frac{truepositives}{truepositives + falsenegatives}$$

A true positive was considered when the network make a detection with the IoU bigger than 50% when compared with ground truth. A false positive is any other detection performed by the network. Finally, a false negative is represented by all the objects that were not detected by the network and were labelled. The best value for the F1-score is 1, with 0 being the worst score possible.

Considering that this metric takes into account all of these measurements, it eliminates the possibility of an horrendous implementation that, for instance, makes a lot of wrong detections, something that is not considered by the mAP metric. For this reason, both of these metrics will be used to attest the correct implementation of the object detection networks.

4

NEURAL NETWORK INFERENCE SYSTEMS

Since one of the main goals of this work was to determine the most efficient inference strategy for the embedded device, the first step was to test the already available frameworks. Interestingly, all of them use the same library to interface with the GPU (cuDNN) but deliver very different performance results. It became apparent that this library was implemented differently between frameworks. For this reason, it became imperative to develop and test a deployment framework that used the same back-end but fine-tuned to better fit the embedded device's characteristics. These preexisting tools were also useful to establish a baseline to determine how the developed framework fairs in comparison with already developed methods, providing relevant insight to determine if its development is justifiable.

These next sections introduces the studied neural network libraries and frameworks, the reason they were considered and their pros and cons. It is also pertinent to mention that the absence of an *OpenCL* based library is related to the fact that NVidia does not support it in the embedded device under testing. This is a major disadvantage of this system since it forces the use of CUDA to take advantage of the GPU, making a possible platform change in the future dependent on NVidia solutions.

4.1 NEURAL NETWORK LIBRARIES

The primary focus of a low level neural network library is, obviously, performance. The most common GPU oriented libraries are cuDNN and TensorRT, even though these are closed source and support NVidia hardware only. In particular, cuDNN is the backbone of many neural network frameworks due to its ease of use, the proliferation of NVidia hardware with great performance and its efficiency. TensorRT is less well known simply because it is much more focused in deployment, leaving out training routines. It is also advised to use it alone, instead of integrating it with a neural network framework, since it already implements automated ways of translating models from other libraries (like caffe and Tensorflow). Unfortunately, this automated process does not play well when the network contains uncommon layers.

4.1.1 *cuDNN*

cuDNN is a library that implements neural network primitives efficiently for NVidia GPUs Chetlur et al. (2014). It is one of the most used neural network libraries for GPU support (even though it is only compatible with NVidia hardware), providing the GPU backend for Tensorflow, *Theano*, Caffe, PyTorch, among others. It is sometimes described as the BLAS library for neural networks on GPU since in enables the same level of performance, maintainability and versatility that BLAS enables for General matrix multiply (GEMM) operations. Being developed by NVidia, it is expected that the primitives implemented take full advantage of the exceptional parallel capabilities of their GPUs, outperforming any third party OpenCL or CUDA implementations.

This library contains not only primitives for inference but also for training, making it a very complete solution if there is a need to add GPU support to a neural network framework. It is also ready to take advantage of the new tensor cores that equip the Volta architecture Markidis et al. (2018) and supports FP16 and INT8 operations (for the hardware that supports it).

Programming model

cuDNN programming model adopts a very typical programming model for a high performance library. A handle has to be initialised and passed to every subsequent library function call. This handle specifies the GPU and CUDA streams to use with the respective calls.

Being a neural network library, there is obviously a need to manage large quantities of memory. Unfortunately, this task is not eased by the library since it is completely detached from it. Memory allocation/deallocation has to be performed manually by the memory management functions available in the CUDA toolkit. This has the advantage of allowing the developer to decide how to allocate memory, which, for this work, proved to be essential to experiment with `cudaMallocManaged()`, a function that allocates memory that is accessible by the CPU and the GPU with no explicit memory copies.

A typical neural network layer function call has to be preceded by:

- cuDNN handle creation;
- Input/output tensors specification (shape, data type, memory organisation);
- Tensor memory allocation and filling.

After all these steps, some functions have a few implementation algorithms that the developer can choose from. Some of these algorithms need additional workspace memory, which creates the need to call an additional function to determine the size of the respective

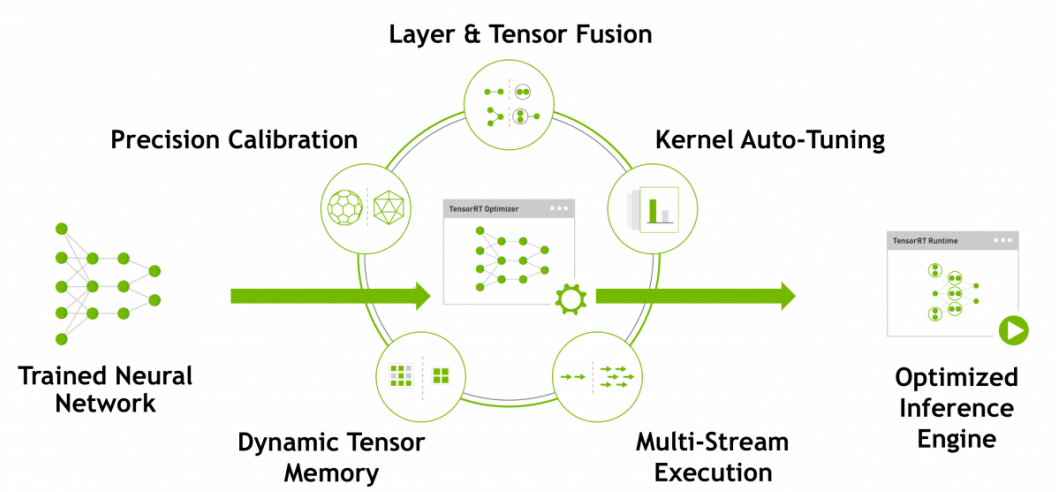


Figure 19: TensorRT workflow (Adapted from NVidia (2016))

workspace memory for the respective algorithm. These multiples implementations proved vital since great performance gains were achieved by changing the default algorithms.

4.1.2 *TensorRT*

TensorRT is a neural network framework specifically designed for deployment purposes by NVidia. It distinguishes from cuDNN since it is only capable of inference and performs model optimisation, something that cuDNN is not capable of. Among the optimisations are layer and tensor fusion, precision calibration and kernel auto-tuning [1]. These optimisations may result in very minute discrepancies at the end result when compared with a cuDNN implementation, but these should not impact the overall network performance metrics. Like cuDNN, TensorRT is fully capable of taking advantage of the newly available tensor cores and can perform most common neural network operations in half precision.

Programming model

The programming model implemented by TensorRT is much more automated and intuitive than cuDNN. Since the goal here was to develop a high performance inference only library, all the memory management and optimisations are automatically performed.

Figure 19 perfectly portrays the workflow of a TensorRT based application. The library receives a pre-trained neural network model from other frameworks. This step can be performed automatically if the model comes serialised with certain specific formats (e.g.:

1 As stated by NVidia in https://developer.nvidia.com/TensorRT

Caffe2). When the serialisation format is not supported, the library has the option to create the model, layer by layer, programmatically.

After the model is transmitted to the library, it starts an automatic optimisation process. Being a closed source library, the specifics of what it does are unknown. After this step, the inference engine is created and is ready to infer on a given input.

This optimisation step is relatively long and is not suitable to be performed on a live system. For this reason, the optimised inference engine can be serialised to a *PLAN* file, making this lengthy optimisation step unnecessary.

4.2 Neural network frameworks

Neural network frameworks are the interface that the developer uses to implement, test and fine-tune ML models. They create an easy to use development environment that is normally agnostic to system and sometimes is even agnostic to its back-end, making it possible to run them, efficiently, on CPUs and GPUs of multiple vendors.

The next few sections presents the neural network frameworks that were tested on the platform. They were chosen based on relevance for the main project (Darknet), popularity (Tensorflow) and claimed ease of deployment (PyTorch).

For each framework, easy of installation and use, flexibility and overview of the implementation is also included in the presentation.

4.2.1 *Darknet*

Darknet is an open source neural network framework developed by Joseph Redmon in C and CUDA Redmon (2013–2016). It is one of the most easy to use and install frameworks due to its simplicity. It implements the most common neural network layers in both the CPU and the GPU (NVidia only) in a very linear and easy to understand way.

This framework was especially relevant for the project for three main reasons:

- it is the framework in which YoloV3 was initially developed;
- the implementation is easy to understand, although poorly documented;
- it is very easy to compile and execute on Jetson TX2 with GPU support;
- it is one of the frameworks used in the main project.

Due to its simplicity and the fact that is was already being used during the main project development, this framework will be studied in more detail since it provided the model specification and weights formats for the deployment framework developed. Adopting this

serialisation format turns Deeploy into a drop-in replacement for Darknet, enabling development work to be quickly and efficiently deployed. It also simplified and quickened the development process, since another neural network serialisation format was not developed.

The Darknet model serialisation format clearly separates the model specification from its weights by storing them in different files with different formats: `CFG` format for model specification and `Weight` files the layers weights. The next section presents each format individually.

`CFG` file format

The `CFG` file is very similar to an `INI` file since it is also composed of sections, properties and values. The file format starts with a `[NET]` section that specifies the input shape of the network, batch size and other training parameters. Most of this information can be safely ignored for inference purposes.

The next sections on the file define, one by one, the layers that define the network. The section name defines the layer type and the property/value pairs define the layers specification. Here are three examples of sections that compose a `CFG` file:

```
[net]
batch=1
subdivisions=4
height=256
width=256
channels=3
learning_rate=0.00001
momentum=0.9
decay=0.0005
```

```
[convolutional]
filters=64
size=3
stride=1
pad=1
activation=relu
```

```
[route]
layers = -1, 61
```

Figure 20: Example sections from CFG format.

It is also important to note that the input of every layer is the preceding layer, except for the `route` and `shortcut` layer in which the specific input layers have to be specified.

This format produces a very easy to read and manipulate configuration file for model specification. With this file alone the framework is capable of allocating the necessary memory for the model and, when provided with a dataset, train it.

Weights file format

The weight file format is extremely basic and limited. It is composed of a small 20 byte header that specifies the Darknet version followed by a binary dump of all the weights by the order their respective layers appear in the CFG file. It does not contain the layer type,

the weights size or even the number of layer, rendering the weights file useless when not combined with the correct CFG file.

Darknet implementation overview

Darknet has three distinct implementations of each supported layer type.

- **CPU**: A complete but inefficient CPU implementation that is only viable as a debug mechanism. This implementation does not use any third party libraries (except for openMP for multicore support) resulting in a very slow implementation. The implementation even uses its own methods for GEMM operations.
- **GPU (CUDA)**: Although inefficient, this implementation is much faster than the CPU implementation (for a balanced system) since it makes use of the GPU. Here, every layer was written directly in CUDA, making use of the an NVidia GPU but in an inefficient way. There was not a big focus in writing efficient implementation, the focus was more gear towards code legibility and maintainability.
- **GPU (cuDNN)**: This is the fastest implementation. It makes use of the function provided by NVidia's cuDNN library. This library is a very efficient neural network library that implements most of the common layers types and support forward and backward propagation (inference and training respectively). Being a library written by NVidia for its own hardware, it is extremely fast and takes full advantage of the GPU computational power.

Lastly, contrasting with the other frameworks of this chapter, it is only possible to interact with Darknet via command line arguments and input files (`CFG` and `weights`). This obviously leads to a less flexible framework but simplifies development and testing of new neural networks.

4.2.2 *Tensorflow*

Tensorflow Abadi et al. (2016) is the most popular neural network framework by most metrics: web searches, repository activity or references. This makes it extremely relevant to compare against, since it provides a strong reference point for performance comparisons. It is currently open source and actively developed and used by Google.

One of the main features of Tensorflow is the promise of a very flexible and portable numerical computation framework based on flow graphs. These graphs are the most fundamental design principal behind this tool. The framework works by initially defining the set of operations in a static flow graph (with vertices being the operations and edges the

tensors) and passing this graph to the engine. This turns the framework particularly efficient since it can analyse and optimise the graph before execution. This also negates all the impact an interpreted language like Python might have in its performance.

The scripting language is essentially only used to describe the computational graph, with the execution itself happening on a more efficient environment (namely high performance C++ libraries in the case of Tensorflow). Figure 21 illustrates the computation graph (generated with *Tensorboard*) resulting from the example code.

```
import tensorflow as tf

with tf.Session() as sess:
  m1 = tf.constant([[1, 2], [3, 4]], name = 'm1')
  m2 = tf.constant([[3, 4], [5, 6]], name = 'm2')
  m3 = tf.constant([[5, 6], [7, 8]], name = 'm3')

  a1 = tf.add(m3, m2, name = 'Add')
  p1 = tf.matmul(m1, a1, name = 'Mult')

  result = sess.run(p1)
```

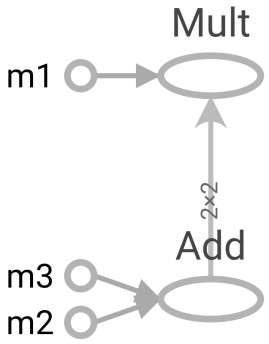


Figure 21: Example Tensorflow code and its resulting computation graph

The depicted code is a valid Python program that uses Tensorflow to perform 2 matrix operations: an addition and a multiplication. The standard mechanism for data movement in this framework is the tensor which can be described as multi-dimensional arrays. Another important characteristics of the tensor is that it can be stored on the CPU or the GPU memory. In fact, in the same graph, multiple tensors can be stored in different devices, with the necessary data movements being managed, automatically, by the framework. This simple design principal allows Tensorflow to more efficiently take advantage of all the hardware available in the system. GPU support is implemented via the CUDA toolkit, limiting it to NVidia hardware. Excluding GPU support, Tensorflow is extremely flexible since it is open source, allowing it to be compiled in most devices.

4.2.3 *PyTorch*

PyTorch Collobert et al. (2011) is an open-source Python library developed by Facebook's AI Research Group (FAIR) for Machine Learning. It is based on Torch Collobert et al. (2002) and distinguishes from Tensorflow since it is imperative. In this context, being imperative means that the user does not describe a static computational graph that gets passed to an execution engine. Here, the operations described are immediately performed, greatly simplifying debug and allowing for dynamic neural networks Lorica (2017).

Most of the vocabulary used in torch is common among neural network libraries, with the main components being `Tensor`, `Function` and `Module`. The first, like previously, can be interpreted as a multi-dimensional array. These data structures are the main data moving method in most neural network libraries. `Functions` are operations that can be performed on a given input, e.g.: the `log()` function. These operation are completely predictable and cannot have internal state like weights. An operation of that nature, in torch, is called a `module`. Examples of modules are convolutions or fully-connected layers. Figure 22 exemplifies the PyTorch version of the matrix operations previously implemented in Tensorflow (in figure 21).

```
import torch

m1 = torch.Tensor([[1, 2], [3, 4]])
m2 = torch.Tensor([[3, 4], [5, 6]])
m3 = torch.Tensor([[5, 6], [7, 8]])

a1 = m3 + m2
p1 = torch.mm(m1, a1)
```

Figure 22: Example pytorch code

With this example the differences between the frameworks become clearer. Due to its imperative architecture, PyTorch does not need a two stage implementation where a description of the desired computation is done followed by the computation itself. Here, every operation happens at the moment of its function call.

Like most neural network libraries, PyTorch is also capable of taking advantage of both the CPU and the GPU, being completely dependant on `cuDNN` to support the latter.

5

THE DEEPLOY TOOL

The Deeploy tool is the main implementation effort of this dissertation. It served a dual purpose: a high performance and flexible test environment for TensorRT and cuDNN and a prototype inference tool for Darknet developed models for Jetson TX2. The goal for this deployment/test environment was to determine the most efficient way to deploy a DNN model on the embedded device with no changes to the network architecture (other than the ones performed by TensorRT or changes in precision of floating point operations). It was also important to not interfere with the already established development environment and methodologies.

It was imperative to understand the already established development methodologies and acquired hardware to develop a robust and useful deployment strategy. The adopted process resembles *CRISP-DM* Chapman et al. (2000), depicted in figure 23, with all the model development stages, from Data Understanding to Evaluation, being performed in dedicated workstations, composed of consumer grade Pascal GPUs.

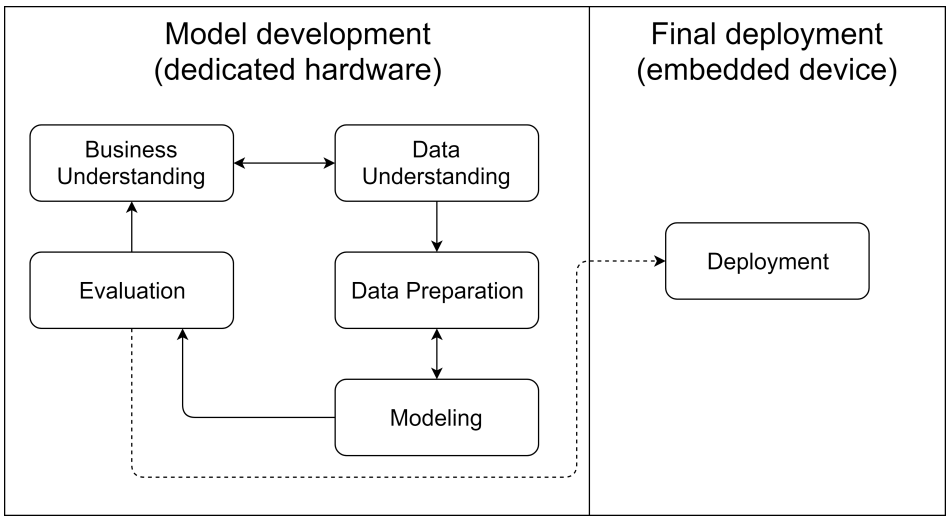


Figure 23: Employed development workflow

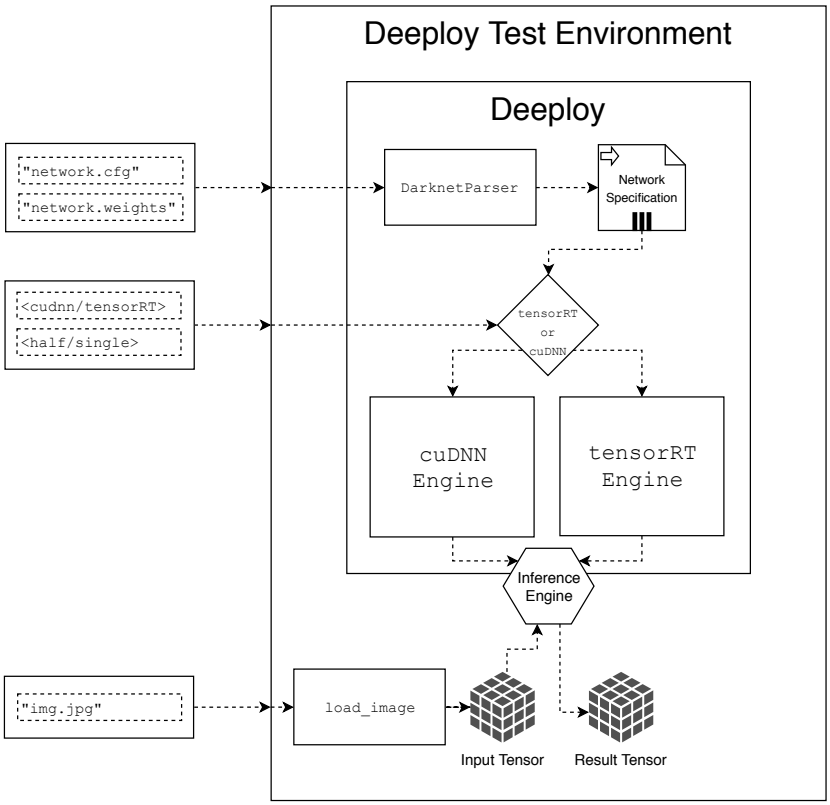


Figure 24: Deeploy Architecture

As stated in previous chapters, these devices have a very appealing single precision floating point operation throughput, but very limited half precision capabilities, contrary to the embedded device used in deployment. To solve this issue, Deeploy only intervenes in the deployment stage and enables the possibility of inference occurring at half precision while training is performed in single precision. This has the potential to greatly decrease inference times, but it also opens the door to model performance degradation since small rounding error get propagated along the network. Deeploy addresses this by allowing testing of the model against a ground-truth dataset to compare metrics between floating precisions.

Since the main project is not yet in the deployment stage, it was necessary to also develop a test environment to validate the tool. The test environment depicted in figure 24 uses Deeploy to perform inference on a given input image, with various network models and respective weights. It can also use both Deeploy inference engines (cuDNN and TensorRT) with both floating point precisions (depending on the input arguments).

This environment enables testing against reference results on realistic workloads. The programming language chosen for the project was C++ due to its higher performance, compatibility with the TensorRT API (also written in C++) and its higher versatility when dealing with objects and memory management (compared to C).

The same diagram also portraits the intended use for the tool: to create an inference engine. Deeploy is able to perform this task when provided with the paths for the network configuration and its weights, the intended back-end to use (cuDNN or TensorRT) and the desired floating point precision. Another setting that can be tuned is the batch size, not depicted in the diagram.

Analysing now the graph with the Deeploy in figure 25, it is possible to see that Deeploy in figure starts by parsing the network input files in a class named `DarknetParser`. As the name implies, this class is responsible for parsing, layer by layer, the network and its weights. This generates a data object of type `NetworkSpecification` that holds all the necessary information for the engine creation. This object is passed to the correct engine class with the necessary arguments (batch size and floating point precision).

After the desired engine is loaded and successfully created, inference can be performed. In the case of this environment, inference happens over the input image that was previously loaded into a Tensor. Similarly to other frameworks, a `Tensor` class was developed to facilitate data movements and, in the case of this framework, floating point precision conversion.

All of these components are presented in more detail in the following sections.

5.1 TENSOR

The `Tensor` class is the first component to be presented since it is the only that has no dependencies. Similarly to other framework, this class is responsible for managing multi-dimensional arrays. It is used for every memory management operation related to the inference engines, like memory allocation for weight loading, input and output during inference operations and memory freeing when it is no longer needed.

Encapsulating the complexity of memory management in a class was necessary for this project since memory movement operations happen throughout every stage of the inference process (from loading to actual inference) in multiple classes. This way, the complexity of memory management operations is hidden and code duplication is avoided.

Its actual implementation is performed using only 4 internal variables (as described in figure 25). These variable help the class to keep track of location of the current tensor (`CPU` or `GPU`), its current type (half or single precision), shape and its pointer. The most important function to interface with this class (apart from its constructors) are `setCPU()/setGPU()` and `setType<float>()/setType<__fp16>()`. These functions, as their names suggest, force the Tensor to change memory location or floating point precision, respectively. Their return type (`&Tensor`) allows method chaining, resulting in very intuitive manipulation (e.g.: `Tensor.setGPU().setType<__fp16>()` will, if necessary, move the tensor to the GPU and

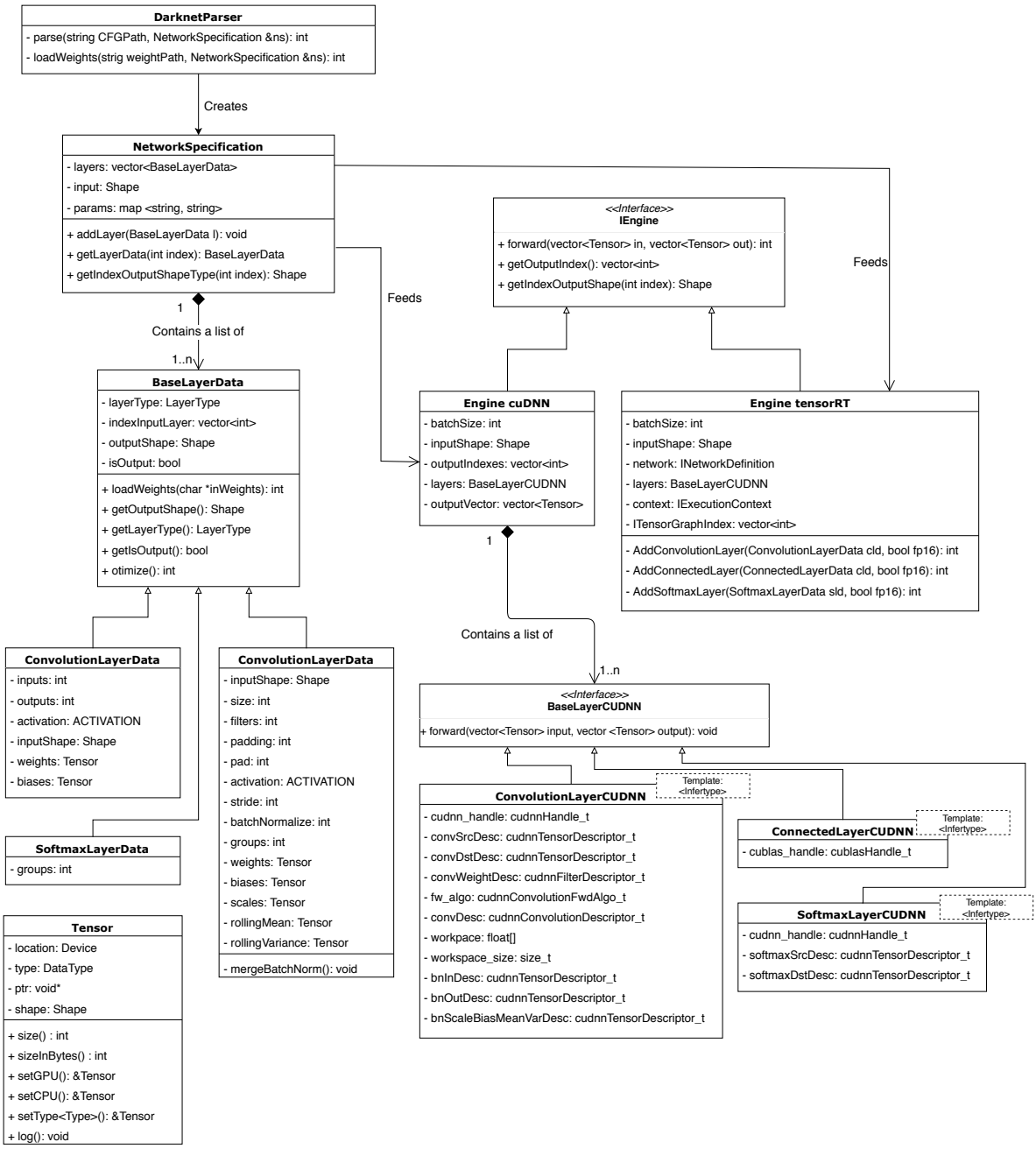


Figure 25: Simplified Deeploy class diagram

convert each of its elements to half precision). It is also worth noting that all the combinations of data movements and type casting are supported.

Finally, this class implements all the necessary constructors and destructors, leaving the burden of memory management to the compiler, even when dealing with allocations on the GPU.

5.2 DARKNETPARSER

This class is responsible for loading the `CFG` and the Weights file to memory, creating the `NetworkSpecification` object. This design decision makes the project more versatile, allowing for a possible addition of new parsers for other network serialisation formats.

The parser itself makes use of `regex` to detect all of the sections of the CFG file (figure 20) and its respective property-value pairs. The pairs are stored in a `map<string, string>` that gets passed to the constructor of the respective layer type. This `map` contains the settings for the layer and is consumed by its constructor with the help of an auxiliary parsing class (not depicted in the class diagram). After layer parsing, the layer is inserted into the `NetworkSpecification` object via the `addLayer()` method.

For `weight` loading, the parent class function `loadWeights` is called for each layer after loading the whole file into memory. Due to the very simplistic nature of the file format (sequential binary dump of weights of all layers), this function needs to return the amount of data read from the input pointer so that the next layer knows from which memory address to start to read weights from, hence the `int` return type.

5.3 NETWORKSPECIFICATION

This class is responsible for representing the network and its weights in memory, layer by layer. It essentially stores, sequentially, a list of `BaseLayerData` derived objects.

Here the first set of optimisations happen with the `optimize` function call. This function should, as the name implies, optimise the network in a way that does not alter its output, but enhances its inference performance. Currently, the only optimisation performed at this stage is the fusion between batch normalisation and convolutional operation, on the convolutional layer. This is performed by calling, for every layer, the `optimize()` method that is implemented by every layer type, even though only convolutional layers benefit from it. This optimisation stage can, in the future, perform more advanced optimisations like the removal of redundant layer or the fusion of other layer types.

5.3.1 *Batch normalisation and convolution fusion*

Referring back to the batch normalisation formula in figure 10 and to the convolution layer explanation in figure 8, it is possible to prove, with some mathematical manipulation that these layers can be merged by applying the following bias/weight transformation:

$$B_f = \frac{\gamma_f * (B_f + \mu_f)}{\sqrt{\sigma_f^2 + \epsilon}} \text{for } f = 1, ..., F$$

$$W_{fchw} = W_{fchw} * \frac{\gamma_f}{\sqrt{\sigma_f^2 + \epsilon}} \text{for } f = 1, ..., F, c = 1, ..., C, h = 1, ..., H, w = 1, ..., W$$

For a convolution with F filters, an input tensor with C channels and a convolution size of H by W.

This optimisation not only reduces the amount of computation performed at inference time, but also reduces the number of kernel calls. It is also important for TensorRT testing since this framework does not support this type of layer.

5.4 cuDNN based engine

For both engines, there are two important phases to describe: engine creation and inference. cuDNN follows a very typical pattern for a high performance numerical computation library. It provides a set of high performance, asynchronous functions that accomplish a typical Deep Neural Network operation like convolution, activation or batch normalisation. This programming model creates a very demanding development environment with memory management, function call, device synchronisation and weight loading being coordinated by the developer.

In the class diagram of figure 25, it is possible to attest the amount of necessary variables needed perform a convolution operation with cuDNN, along with 2 additional simpler examples. Worth noting that the class diagram does not contain all the implemented layer for space constraint reasons. The `cudnnTensorDescriptor` data-type variables, like the name implies, describe a tensor, from its shape to its element type. The other variables describe the algorithm to be used during forward propagation (`fw_algo`), the overall settings for the convolution (`convDesc`) and the worskpace allocated and its size (`workspace` and `workspace_size`, respectively). These latter two variable were only necessary on the convolution operation, since it is the only operation that benefits from a workspace during computation. All these variables are then passed to the respective cuDNN forward function, at inference time.

Figure 26 displays the construction phase and the forward phase graphically. During the first phase, the engine creates the necessary cuDNN handle, then, for each layer in the inputted `NetworkSpecification` object, the respective layer constructor is called. All of the `CUDNN` layer objects inherent from the `BaseLayerCUDNN` class, forcing the existence of the `forward()` method, that is called during inference for each layer, simplifying the inference execution. The first three steps inside the loop happen inside the layer's constructor and are generally applicable to all layers. At the end of each iteration, it is necessary to store the created layer in a array of `BaseLayerCUDNN`. Finally, with all the layers created, allocation of the necessary space for the output of each layer is performed.

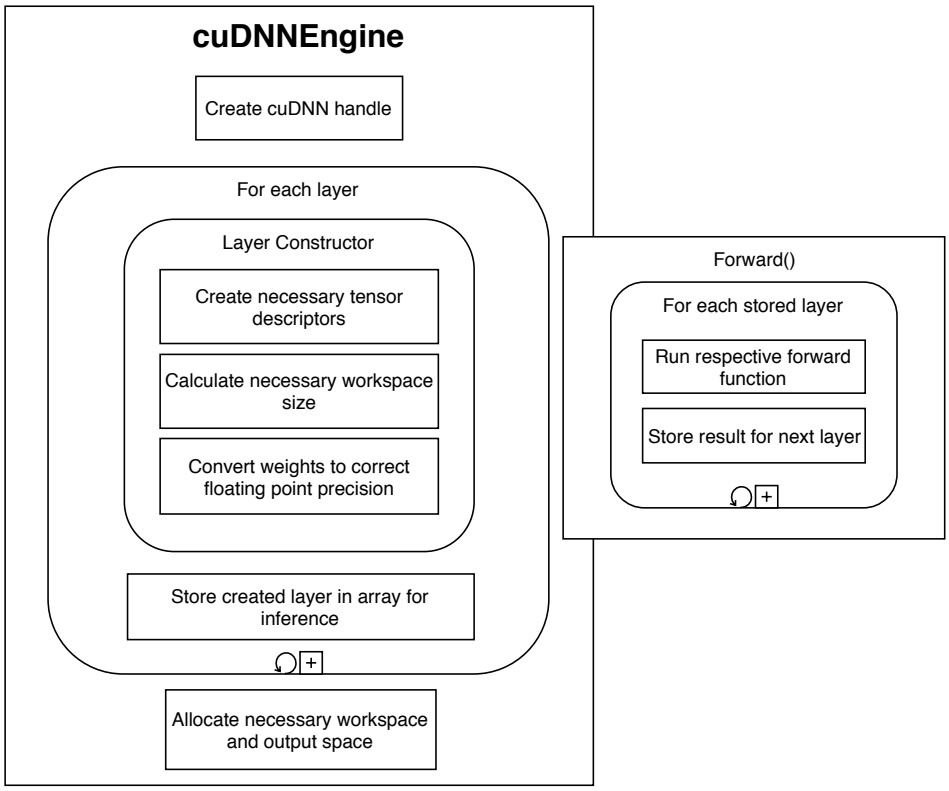


Figure 26: Simplified cuDNN engine construction and forward phases

The `forward()` procedure is less complex. Essentially, for each previously initialised and stored layer, is it necessary to call its `forward()` method. It is important to mention that, as expected, the forward method of a layer can only be executed after the `forward()` of all of its preceding layers.

5.5 TensorRT based engine

TensorRT, as stated by NVidia, is an inference only deployment focused library for ML models. It is much less flexible than cuDNN, handling all of the memory management,

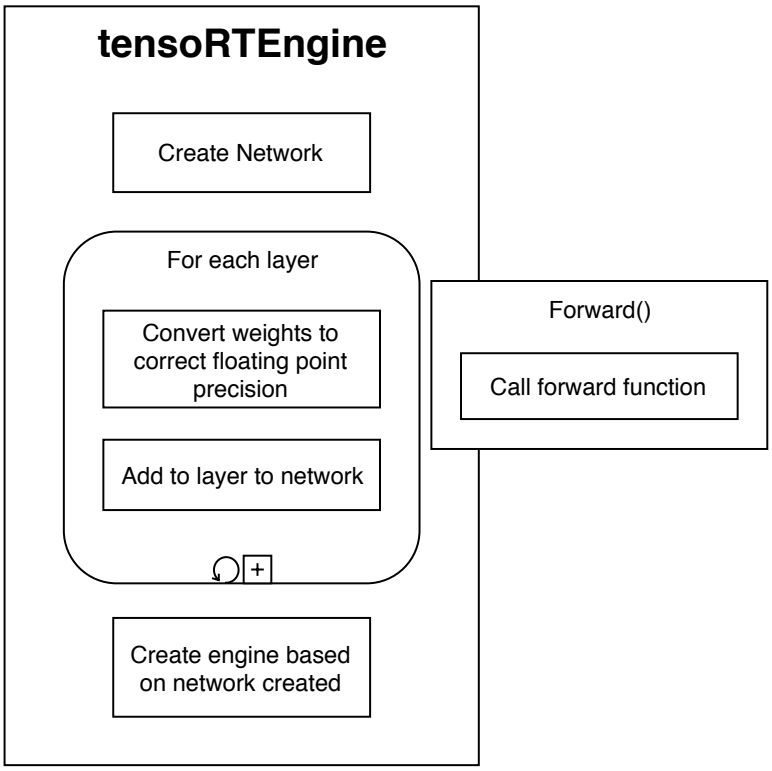


Figure 27: Simplified TensorRT engine construction and forward phases

weight loading and kernel launching on its own. The way to use TensorRT, is to describe the network and then wait for the library to perform its optimisations. This approach creates much easier to maintain and read code, but actually greatly complicates debug operations.

Figure 27 clearly illustrates the simplicity of creating a TensoRT based engine. In the constructor, there is only a need to create a network object (`nvinfer1::INetworkDefinition`) and, for each layer in the input `NetworkSpecification` object, call the corresponding adder method, after the necessary floating point precision conversion of the weights, if necessary. This is then followed by a function call that automatically optimises the network and generates the inference engine. It is important to note that, this function call takes a few minutes to execute and, for this reason, should be avoided. For the forward phase, like depicted, it is simply necessary to call the forward function provided by the library.

The problem with this completely integrated approach comes when there is a need to use exotic or less common layers. NVidia answers this problem by creating a plugin interface that essentially allows the developer to temporally exit TensorRT's optimised run-time and execute custom code. This completely mitigates the previous issue from a practical view point, but might create performance issues.

6

VALIDATION AND PERFORMANCE ANALYSIS

After the initial implementation, a profiling and fine-tuning step is mandatory as well as a validation stage. This enables the detection (and possible elimination) of bottlenecks in the execution environment. Here the main focus will the cuDNN and TensorRT based engines, but testing will also be performed on the Unified Memory programming paradigm provided by NVidia. Finally, a validation step is performed to ensure the end results are correct.

6.1 PROFILING AND FINE-TUNING

Profiling was performed remotely via NVidia Visual Profiler (NVVP) with embedded device in its highest performance mode. The version of JetPack (Jetson's OS) was 3.3, the latest version at the time of writing. This version comes with CUDA 9.0 pre-installed as well as cuDNN (version 7.1.5) and TensorRT (version 4.0). The model chosen to be analysed during profiling was YoloV3 mainly due to its importance to the project but also due to its more demanding characteristics (when compared to the other network architectures).

6.1.1 *cuDNN based engine*

The profiling information (depicted in figure 28) clearly accentuates the need to focus on the convolutional layer. This is corroborated by the time spent on the `maxwell_cudnn_128x128` kernel, used during forward pass of the this layer type. The inference times, for one single image, were 139 and 221 milliseconds for half and single precision, respectively. These inference times also emphasise the much higher efficiency of this embedded device when dealing with half-precision floating point computations.

Convolution layer

The convolution forward function defined by cuDNN, allows the specification of the forward algorithm to use. This argument (of type `cudnnConvolutionFwdAlgo_t`) can specify

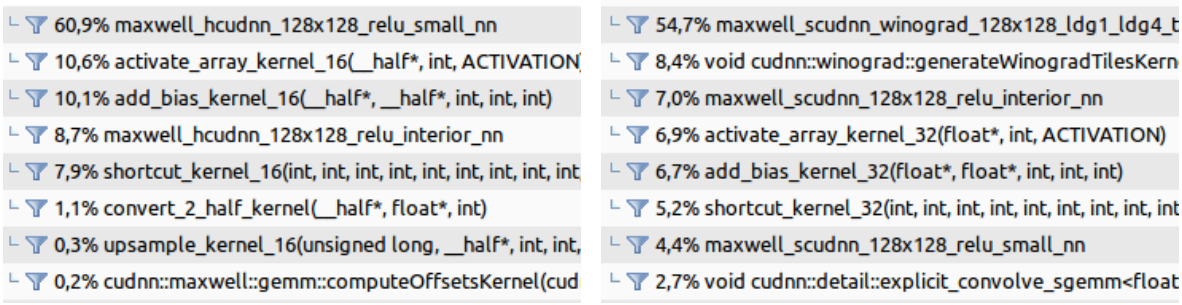


Figure 28: Relative time spent per kernel in half-precision (left) and single-precision (right) inferences on cuDNN based engine

Algorithm	Half-Precision (ms)	Single-Precision (ms)
Automatic Algorithm Selection	**139,44**	221,23
`ALGO_IMPLICIT_GEMM`	197,17	335,54
`ALGO_IMPLICIT_PRECOMP_GEMM`	140,06	**193,25**
`ALGO_GEMM`	ERR	273,06
`ALGO_DIRECT`	ERR	ERR
`ALGO_FFT`	OOM	ERR
`ALGO_FFT_TILING`	OOM	ERR
`ALGO_WINOGRAD`	ERR	ERR
`ALGO_WINOGRAD_NONFUSED`	ERR	ERR

Table 8: Inference times for each convolution forward algorithm

up to 7 distinct convolution forward algorithms. The current algorithm choice was made by a function (`cudnnGetConvolutionForwardAlgorithm()`) also available in the cuDNN toolkit that serves as a heuristic for obtaining the best suited algorithm for forward convolution, given some conditions. Testing each forward algorithm individually yielded the following results:

Interestingly, even when specifying that the main criteria was speed, the automatic function kept choosing sub-optimal algorithms for single precision inference, resulting in inefficient results. Clearly, automatic selection is not well implemented for this specific hardware, with a fixed choice in the `ALGO_IMPLICIT_PRECOMP_GEM` algorithm producing better results. The "ERR" and "OOM" keywords were used to specify when a algorithm produced wrong results or when it used more memory then the available, respectively.

6.1.2 *TensorRT based engine*

Being a library more focused in deployment, a higher efficiency is to be expected when compared with cuDNN. These expectations were match as the this initial TensorRT implementation managed to make an inference in a single image in just 96ms and 127ms,

depending on floating precision. These results are much better than the ones achieved by cuDNN.

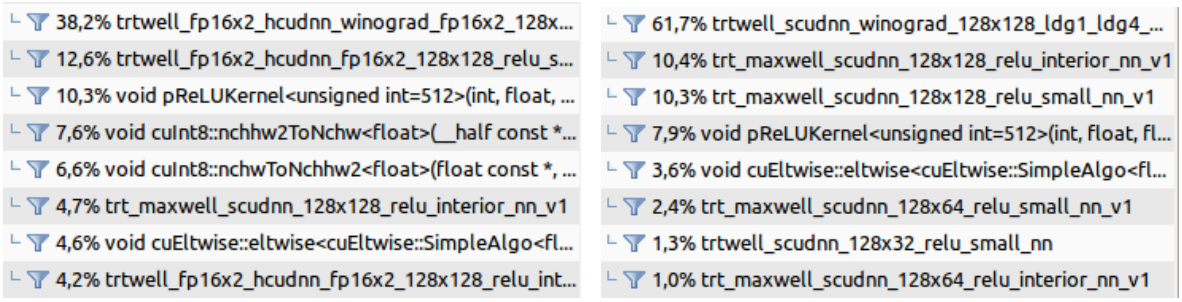


Figure 29: Relative time spent per kernel in half-precision (left) and single-precision (right) inferences on TensorRT based engine

Analysing the profiling results (figure 29), namely the time spent in each kernel, it is possible to detect that, in the half-precision run, a big portion of the time was invested in type conversion. This is due to fact that TensorRT communicates with plugin layers always by single precision, even when in half precision mode. This creates the need to convert floating values twice per plugin layer. This is particularly damaging in this network model case since one of most used activation functions used throughout the model (leaky ReLU) is implemented via the plugin system. The next sections address an way to avoid this problem, by fully implementing these layers with equivalent ones implemented by TensorRT.

Leaky ReLU

Referring back to the leaky ReLU function ($f(x) = max(\alpha x, x)$), it is possible to conclude that this operation can be divided in two phases: multiplication of every element in the tensor by a scalar (α) and the application of the `max` operand between the initial tensor and the scaled one. Even though TensorRT does not directly support the leaky ReLU activation function, it does support a `scale` and a `elementwise` operator. The first is able to multiply an input tensor, element by element, by a factor, while the second can perform basic operations between each element of tensors with the same shape (e.g.: the `sum` or the `max` operation).

Re-implementing leaky ReLU using this method, greatly increases half precision performance due to avoiding unnecessary type conversions.

	YoloV3 - Half-Precision	**YoloV3 - Single-Precision**
Plugin	96,24ms	**127,55ms**
ElementWise/Scale	**78,71ms**	139,25ms

Table 9: Inference times comparison between implementations of leaky ReLU

Analysing table 9, it is possible to notice that this implementation technique further improves the inference times registered under half-precision. In contrast, single precision inference times are hindered. Since there is no need for type conversions on this floating precision, a simple implementation of the activation function via the plugin layer is more efficient then the one using two layers. To achieve the highest possible performance, the best implementation is used depending on the floating precision.

Upsample

The upsample layer was also implemented using the plugin system. To avoid this same type conversion behaviour, this layer was also translated to an TensorRT operation: the deconvolution. This operation, when correclty configured, can scale the dimensions of a tensor, creating the same effect as the original upsample layer. Table 10

	YoloV3 - Half-Precision	**YoloV3 - Single-Precision**
Plugin	78,74ms	**127,67ms**
Upsample	**78,55ms**	128,43ms

Table 10: Inference times comparison between implementations of the upsample layer

In this case, the gains were much more modest. This is simply due to the fact that this layer is only used twice in this model, saving only 4 floating type conversions operations.

6.1.3 *Unified Memory*

Unified memory is a new programming model developed by NVidia that defines a new memory space that all processors see coherently with a common address space. Even though no information is provided on the subject, it is worth exploring the possibility of Jetson, due to its memory configuration, benefiting from this programming model, avoiding unnecessary memory copies.

To perform this test, due to the proliferation of the Tensor class in most memory management operations throughout the project, only this class needed to be changed. The memory allocations were performed by `cudaMemoryMallocManaged()` instead of the usual `cudaMalloc()`, along with other needed adaptations. After these changes, most memory copies were deemed unnecessary and the Tensor class code was much more simplified since there was no need to explicitly handle data movements by calling cudaMemcpy().

It is also important to note that the TensorRT based engine did not work under this new memory management model. Due to the closed source nature of this implementation, it was impossible to determine the cause of the incompatibility.

	YoloV3 - Half-Precision	YoloV3 - Single-Precision
Manual Memory Management	**140,10ms**	**193,70ms**
Unified Memory	167,19ms	221,71ms

Table 11: Manual memory management *vs* unified memory

The results displayed in table 11 are not positive, show an overall hit in performance when when this technology is used. Although much easier to use, this technology is not particularly efficient and does not take advantage of the physical memory configuration present on the embedded device.

6.2 VALIDATION

With the tool fully tuned and implemented, it is now important to validate its outputs to assuring its correctness. Darknet is going to be used as the reference implementation since it and Deeploy share network serialisation formats. This way the same network and weights can be used to compare outputs. This stage also serves to verify the hypotheses that different floating point precisions during the training and testing phases generate a valid inference model.

For the validation stage, the three previously presented network architectures will be used: YoloV3, VGG-16 and ResNet-50. Due to the different types of networks, different metrics have to be used. YoloV3, being an object detection network, is going to be compared across implementations via the achieved mAP. The weights used were the result of training performed by Bosch on a custom vehicle interior dataset with random left behind objects. The validation was performed on a subset of this dataset containing 542 labelled images (figure 30 is a sample image of the dataset).

For the image classification networks (VGG-16 and ResNet-50), 5000 images of the 2012 Imagenet Deng et al. (2009) validation dataset were used. Here, the metrics used were top-1 and top5 error

Table 12 summarises the performance of the YoloV3 model with varying precision and across the three implementations. At single-precision, both Deeploy engines achieve the same results as Darknet. As expected, there was a drop, although small, in both metrics when a lower precision was used. Depending on the needs of the project, the gains in inference time may offset the minimal loss in the detection performance of the model.

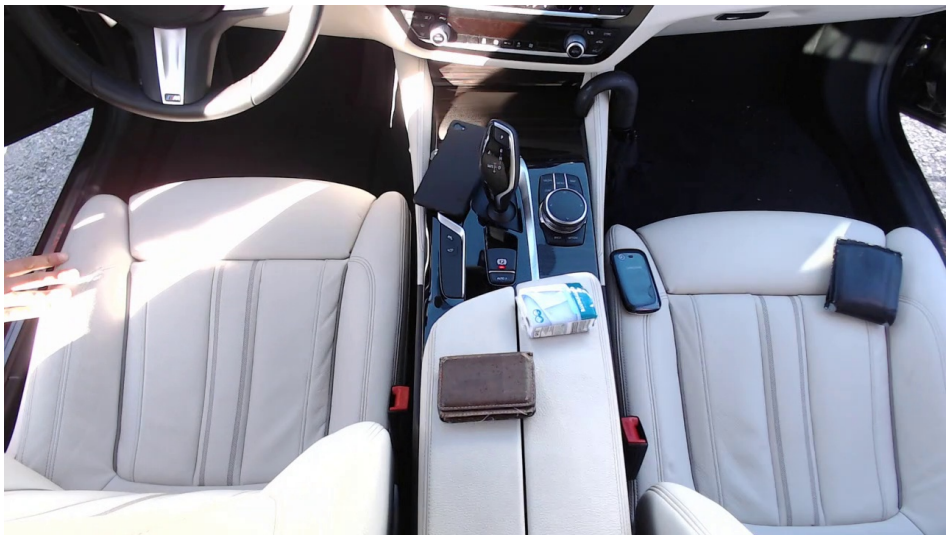


Figure 30: Sample image of the Bosch dataset

	mAP		F1 score	
	FP16	FP32	FP16	FP32
Deeploy (cuDNN)	62,37%	62,56%	0,80	0,81
Deeploy (TensorRT)	62,37%	62,56%	0,80	0,81
Darknet	–	62,56%	–	0,81

Table 12: mAP and F1 of both engines with varying floating point precisions on an internal dataset

Tables 13 and 14 contain the results obtained on the classification networks. All of the results obtained in the VGG-16 network were expected. At single-precision, the error rates achieved by all implementations are exactly the same (29,36% and 10,24% for top-1 and top-5 error, respectively), further proving the correct implementation of both engines. At half-precision there was a slight variation in the final results with a small decrease of top-1 and top-5 error rates.

The same conclusions can be drawn for ResNet-50 apart from a slight deviation in the single-precision top-1 result. Here, the reference framework performed slightly better (25,06% *vs* 25,08%), a discrepancy that can be attributed to different rounding errors produced by the distinct implementations.

	Top-1 error		Top-5 error	
	FP16	FP32	FP16	FP32
Deeploy (cuDNN)	29,30%	29,36%	10,22%	10,24%
Deeploy (TensorRT)	29,32%	29,36%	10,22%	10,24%
Darknet	–	29,36%	–	10,24%

Table 13: Top-1 and top-5 error of both engines with varying floating point precisions on VGG-16

	Top-1 error		Top-5 error	
	FP16	**FP32**	**FP16**	**FP32**
Deeploy (cuDNN)	25,18%	25,08%	7,20%	7,16%
Deeploy (TensorRT)	25,06%	25,08%	7,22%	7,16%
Darknet	–	25,06%	–	7,16%

Table 14: Top-1 and top-5 error of both engines with varying floating point precisions on ResNet-50

Overall, apart from very minute rounding errors, both of the engine created are fully compliant with the Darknet framework. It is also possible to conclude that a reduction in inference precision does not result in a significant model degradation, further approving the strategy of training and deploying at different precisions.

7

FRAMEWORK COMPARISON AND TESTING

With the tool implemented, fully tuned and validated, it is now necessary to test and review the work. This chapter compares the developed prototype against well established neural network frameworks. It starts this comparison by measuring inference times against different tools, followed by a general overview. PyTorch, Tensorflow and Darknet were the chosen frameworks to be used in this comparison, as explained in a previous chapters. Three models were selected to be used: VGG-16, ResNet-50 and finally YoloV3. The first two due to their popularity and the latter due to its role in the main project.

Another performance analysis is also performed, this time, using the various power modes available in the device, making possible to determine how the tool reacts to different clock frequencies and core counts. Here, the tool is also tested in another system, even though it is not one of its use cases. This proves its flexibility and also compares the embedded device to more traditional hardware on deep learning workloads.

7.1 TEST ENVIRONMENT

All tests were performed with the embedded device on the highest power mode (Max-N), except when indicated otherwise. To minimise the normal variation between runs, every test was executed 10 times, after a warm-up execution, and the values shown are the average of these runs. This methodology was chosen since it provides valuable information about the average time the end user will experience while using the final product. Standard deviation was also calculated across the 10 runs and on all tests resulted in values close to 0, highlighting the stability of all implementations. All tests were performed with the Graphical user interface (GUI) on the embedded device turned off to better emulate real world conditions. The software configuration is depicted in table 15.

The workstation mentioned in the table is the same as described in table 4. Like previously, this system will be used to compare the embedded device against common hardware, providing a more intuitive sense of the performance of the device. As standard, only the essential processes were running at the time of the execution of the tests to guarantee reproducible results.

	Jetson TX2		Workstation	
Software	**Version**	**Notes**	**Version**	**Notes**
JetPack	3.3		–	
Ubuntu	–		16.04	
Python	3.5.2		3.5.2	
CUDA	9.0	Latest compatible version	9.0	
GCC	5.5.0		5.5.0	
Clang	–		3.5	Needed to compile Deeploy
cuDNN	7.1.5		7.1.5	
TensorRT	4.0		4.0	
Tensorflow	1.9.0	Installed via wheel file provided by NVidia	1.9.0	Installed with pip3
PyTorch	0.4.1	Compiled from source	0.4.1	Compiled from source
Darknet	508381b	Commit ID no version system	508381b	Commit ID no version system

Table 15: Testing environment

On the embedded device, Tensorflow was installed via a `wheel` package provided by NVidia specifically for the system. On the workstation, this framework was installed via `pip3` utility.

PyTorch was compiled from source directly in both systems. To ensure maximum GPU performance, some source code changes were performed to guarantee that the code was compiled with the correct CUDA compute capabilities version (v6.2) in the Jetson TX2.

Darknet was also compiled from source with the highest performance settings. This means that the project was compiled with the optional cuDNN library and the correct compute capabilities were also specified.

These settings were also used for Deeploy to ensure maximum performance. Due to the proliferation of the half-precision data type (`__fp16`) throughout the project, `clang` had to be used since it can emulate, through software, the needed instructions to handle these types. GCC, in a platform that does not support this format natively, like `x86`, does no compile successfully.

Lastly, it is extremely important to disclose the test methodology for Tensorflow and PyTorch. These frameworks, due to their programmable architecture, are very dependent on the actual implementation used. A framework like Darknet is not as prone to bad results since it is not programmable, having only a command line based interface. To solve this issue, for the most common networks (VGG-16 and ResNet-50), the reference, pre-trained, model provided by the frameworks was used. This way, it is guaranteed that these networks are efficiently implemented and provide a good workload to compare the frameworks. On Tensorflow, package `tensorflow.contrib.slim.nets` was used to test these architectures

while on PyTorch the equivalent package is `torchvision.models`. The most problematic network to implement was YoloV3, as expected. Due to the fact that it is not as popular as the previous architectures, this network does not have a reference implementation on Tensorflow or PyTorch, forcing the use of a custom implementation [1] [2].

Time measurements were dependent on the programming language used. For Python tests, the package `time` was chosen while in C and C++ run time measurements were provided by `sys/time.h` and `std::chrono` respectively. These libraries allow a time resolution of at least 10 microseconds, making them a valid time measurement system for the task.

7.2 FRAMEWORK COMPARISON

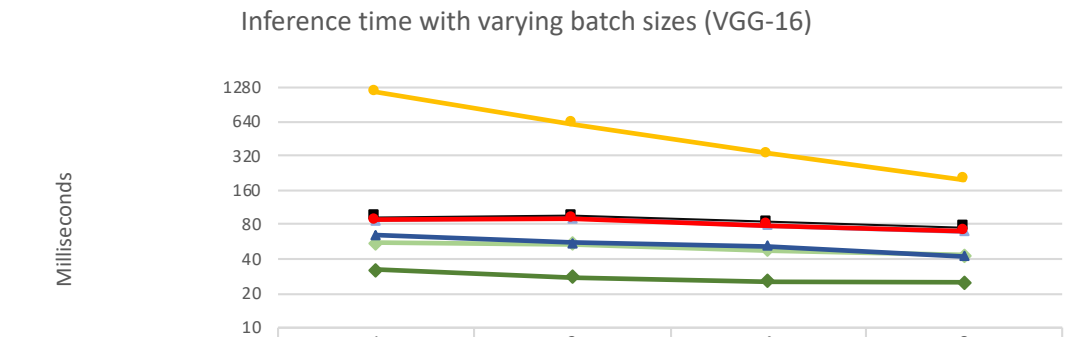
This section will provide a comparison between these frameworks from a performance stand point and will also provide a general overview of the ease of use, support and maintainability of each.

7.2.1 *Performance analysis*

The first network in this comparison is VGG-16 (figure 31). Here, as expected, the highest performance framework was TensorRT (with 25,04ms per inference per image *vs* 42,08ms from cuDNN, the closest framework, with a batch size of 8). This a substantial difference, especially considering that multiple cameras capturing multiple frames is a very likely possibility in the final version of the project. Tensorflow, on the other hand, became much more efficient as the batch size increased (from 1168,68ms per image at batch size 1 to 197,64ms at batch size 8). This shows that this framework was not designed for short, inference only, workloads and is tuned for more demanding workloads on much higher performing devices. The cuDNN based engine (SP), PyTorch and Darknet produced almost the same behaviour which can be explained due to their common back-end.

The relative performance of some frameworks changed when tested with ResNet-50 (figure 32). With this network, Tensorflow behaved better as the batch size increased (from 57,53ms at batch size 1 to 25,26ms at batch size 8). This again is an indicator that this framework is more tuned to bigger, sustained workloads. PyTorch performed exceptionally well considering that its performance rivals the performance of the cuDNN based engine in half-precision (making it about 25% faster when compared with the cuDNN engine at the same precision, at all batch sizes). This is a very unexpected result considering that both engines use cuDNN to perform computations at the GPU. It also indicates that the Deeploy cuDNN implementation has potential to be improved. The rest of the frameworks behaved

1 https://github.com/eriklindernoren/PyTorch-YOLOv3
2 https://github.com/mystic123/tensorflow-yolo-v3



	1	2	4	8
Deeploy (TensorRT - HP)	32.44	27.61	25.53	25.04
Deeploy (TensorRT - SP)	55.83	54.05	47.49	43.56
Deeploy (cuDNN - HP)	65.00	55.89	51.68	42.08
Deeploy (cuDNN - SP)	88.65	92.25	79.91	69.72
Darknet - Single-Precision	92.72	96.29	84.19	75.33
Tensorflow - Single-Precision	1168.58	613.80	338.51	197.64
PyTorch - Single-Precision	88.44	90.37	79.25	70.23

Figure 31: Inference time per image with varying batch sizes (VGG-16)

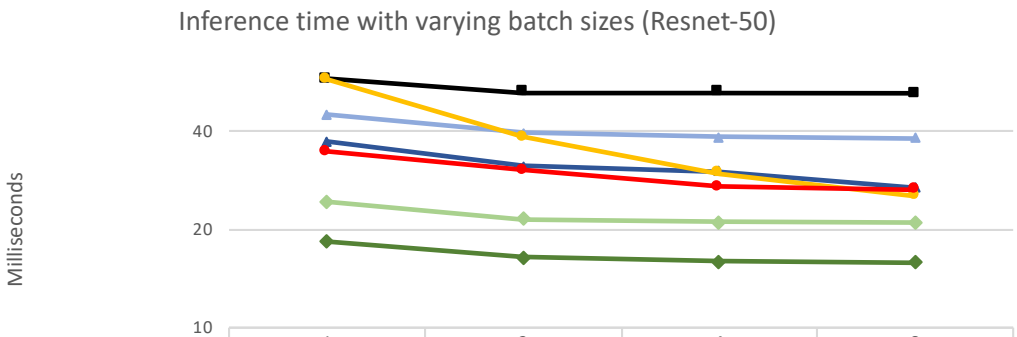
as expected, with TensorRT beating every other implementation (by a substantial margin) and Darknet resulting in the most inefficient inference strategy.

YoloV3, the most computationally expensive network architecture tested and one of the key focus of the main project. Being less popular than the other two network architectures, Tensorflow and PyTorch do not provide a reference implementation. For this reason, these results can be biased depending on the chosen implementation. Nevertheless, the achieved results align with the expectations, with TensorRT performing the best (74,66ms at batch size 8 *vs* 110,34ms from cuDNN, the closest framework). Probably due to the higher computational cost of this network, Tensorflow managed to perform well when compared with PyTorch at lower batch sizes and practically matches its performance at higher ones (168,96ms *vs* 173,47ms at batch size 8). Although Darknet is the reference implementation of this network architecture, it was not particularly efficient with inference times neighbouring the 250ms, while every other framework could produce a result in less than 180ms (at higher batch sizes).

7.2.2 *General overview*

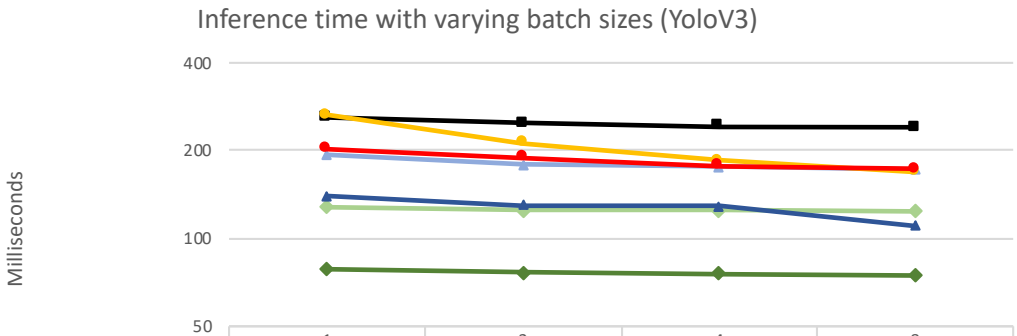
Performance is obviously a very important aspect of the deployment stage but it is not the only one. It is also important to evaluate the suitability of these frameworks on aspects like hardware dependency, model adaptability and ease of use.

Excluding Deeploy, which is still in the prototype stage, Darknet is the most limited tool. Due to its command line interface, it does not support advanced tasks like multi-GPU envi-



	1	2	4	8
Deeploy (TensorRT - HP)	18.35	16.39	15.97	15.80
Deeploy (TensorRT - SP)	24.29	21.47	21.07	20.97
Deeploy (cuDNN - HP)	37.14	31.13	30.12	26.79
Deeploy (cuDNN - SP)	44.91	39.35	38.31	37.91
Darknet - Single-Precision	57.85	52.36	52.34	52.15
Tensorflow - Single-Precision	57.53	38.26	29.60	25.26
PyTorch - Single-Precision	34.64	30.32	26.94	26.41

Figure 32: Inference time per image with varying batch sizes (Resnet-50)



	1	2	4	8
Deeploy (TensorRT - HP)	78.47	76.21	75.66	74.66
Deeploy (TensorRT - SP)	128.01	125.19	124.55	123.85
Deeploy (cuDNN - HP)	139.88	130.11	128.54	110.34
Deeploy (cuDNN - SP)	193.65	178.42	176.48	172.42
Darknet - Single-Precision	259.96	247.63	242.81	240.59
Tensorflow - Single-Precision	265.52	211.64	185.34	168.96
PyTorch - Single-Precision	202.38	188.95	177.84	173.47

Figure 33: Inference time per image with varying batch sizes (YoloV3)

ronments, custom layers or easy integration with other projects. It is clearly a light-weight framework for model tuning and training. On the other hand, it is an extremely simple framework with very little dependencies, making it extremely portable (at least, when compiled with CPU only support). Its GPU support, like every other studied framework, comes from NVidia libraries, limiting it to NVidia hardware. One of the main disadvantages of Darknet is its lack of support and relatively stagnant development, making its adaptability to novel architectures (with new layer types), extremely limited.

Tensorflow and PyTorch can be seen as much more advanced frameworks with better support for exotic layers, distributed systems and can easily integrate with other projects due to their architecture and proliferation. Since most research projects are developed in these tool, their compatibility for novel network architectures are excellent. Like Darknet, both frameworks are dependent on NVidia for GPU support but can be compiled with CPU only support. In terms of performance, PyTorch seems to perform much better than Tensorflow on the embedded device, giving it the advantage. Overall, these frameworks grant the most flexibility when it comes to neural network architectures implementation, but lack in terms of efficiency and hardware support.

Being a tool developed for this specific use case, Deeploy has numerous advantages. The first is its performance with both inference engines. Since these were developed for the given embedded device and for a specific use case, the libraries were more efficiently tuned. Another big advantage is its easy integration with the rest of the main project due to its simplicity and modular architecture. It is also completely compatible with the Darknet serialisation format, making it a drop-in replacement for this framework. Its ability to convert single-precision models into half-precision models should also be mentioned since this greatly increases the efficiency of the inference. The disadvantage of this approach is that, at its current stage, the framework is dependent on the Darknet framework for creating and training new models.

Between both engines, TensorRT is clearly the most appropriate to use in the deployment stage. It is more efficient in every model and, via the `Plugin` interface, can essentially implement any kind of layer, making it extremely versatile. cuDNN should only be considered if the project ever demands training in the embedded device. Here this library can be relevant not only to perform the actual training (since TensorRT does not contain training methods), but also to perform inference while TensorRT engine finishes optimising the model (a process that can take up to 10 minutes, depending on the network size).

7.3 POWER MODES

This last batch of tests aimed to establish a baseline performance for the embedded device at varying power modes, comparing results with the dedicated GPU at the workstation.

The tests were performed at a batch size of 8 with the three selected network architectures on TensorRT and cuDNN at single and half-precision. This batch size better simulates the expected conditions the system will work with at the final project version.

From the TensorRT results, it is possible to conclude that, being performance focused, this tools ignores the hint to perform computations at half-precision in an environment where this would yield poor performance. This is possible to determine due to the very similar results obtained by the tool at different precisions on the consumer GPU. On the embedded device, this tool performs the computation at half-precision, yielding better inference times.

The embedded device in the TensorRT engine is approximately 11x slower than the dedicated GPU on VGG-16 and approximately 8x times slower on both ResNet-50 and YoloV3. These results are mainly due to TensorRT ignoring the instruction to perform all computations in half-precision. At higher precisions, these performance differences increase since the embedded device is approximately half as efficient when performing single-precision floating point computations. This results in it being between 12x and 17x slower when compared with the dedicated GPU, depending on the network.

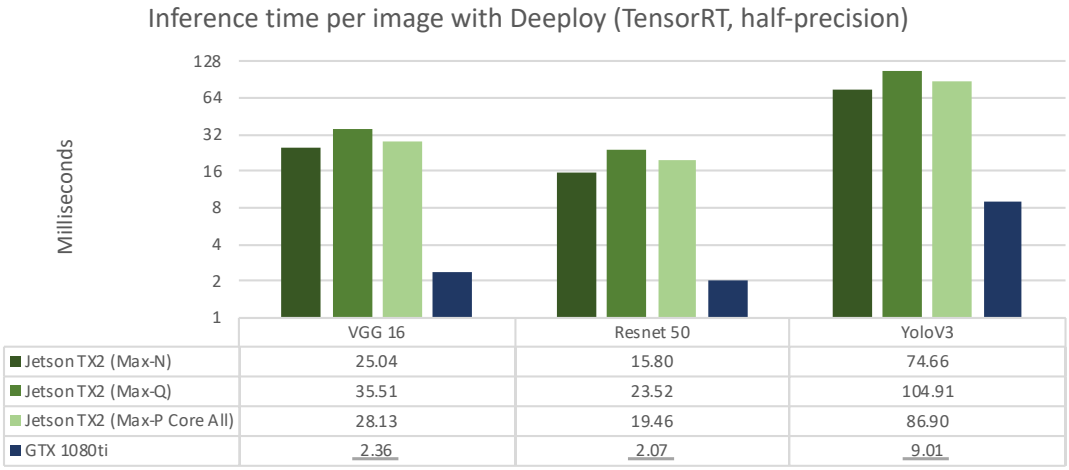
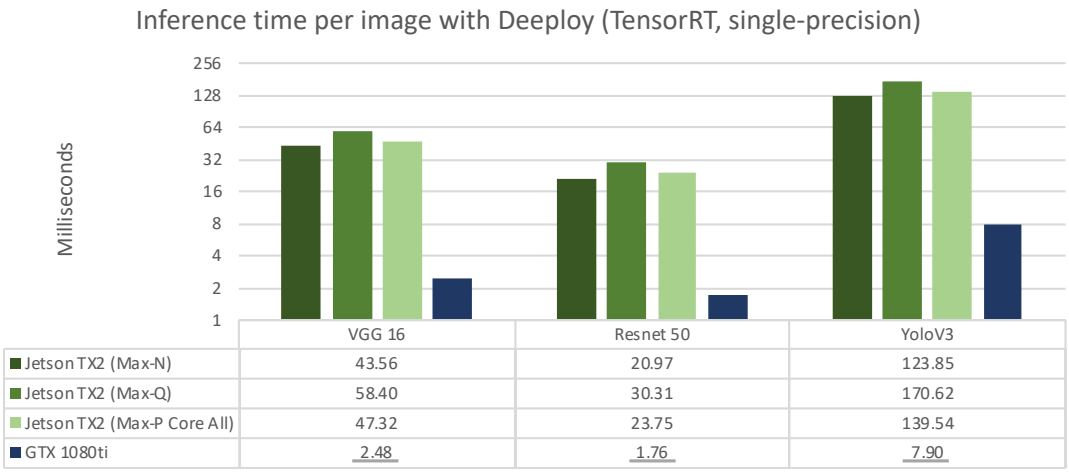


Figure 34: TensorRT inference times with varying power modes in half-precision

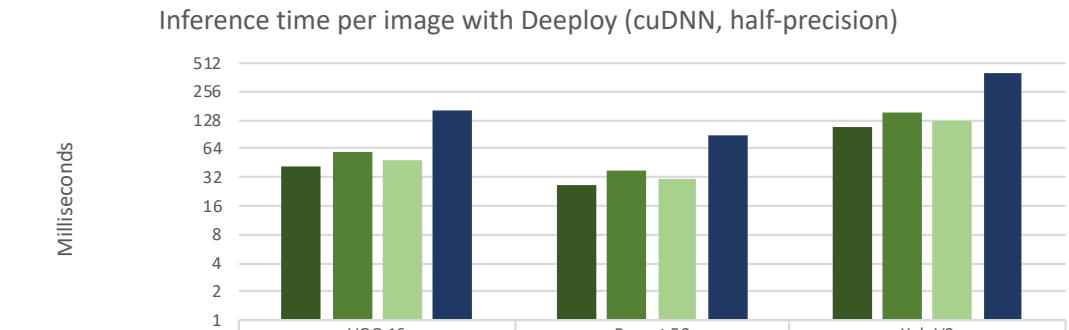


	VGG 16	Resnet 50	YoloV3
Jetson TX2 (Max-N)	43.56	20.97	123.85
Jetson TX2 (Max-Q)	58.40	30.31	170.62
Jetson TX2 (Max-P Core All)	47.32	23.75	139.54
GTX 1080ti	2.48	1.76	7.90

Figure 35: TensorRT inference times with varying power modes in single-precision

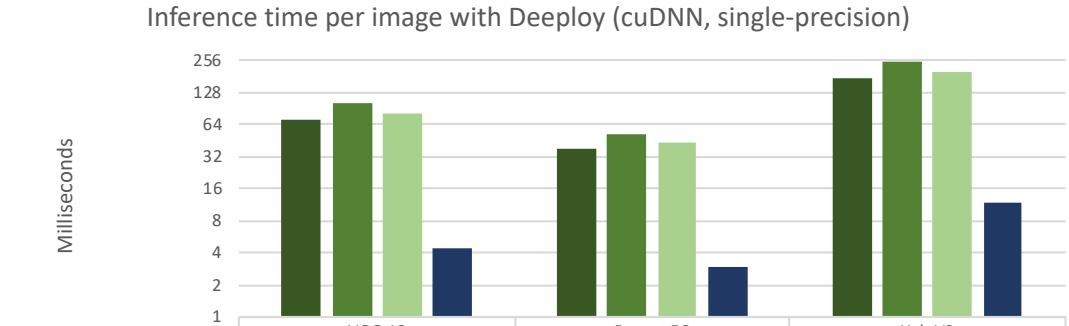
The results achieved in the cuDNN engine were similar to the ones achieved in the matrix multiplication test. Here, the dedicated GPU achieved very poor results when used to perform inference at half-precision, being approximately 3x slower than the embedded device on every network architecture. At single-precision, the dedicated GPU yields roughly the same advantage as the one registered in the TensorRT based engine at the same precision (between 13x and 16x faster).

For both engines, the results obtained in different power modes are to be expected considering that most deep learning inference algorithms are compute bound, making the inference times inversely proportional to GPU core clock speed variations.



	VGG 16	Resnet 50	YoloV3
Jetson TX2 (Max-N)	42.08	26.79	110.34
Jetson TX2 (Max-Q)	60.32	37.83	157.89
Jetson TX2 (Max-P Core All)	47.82	30.39	125.34
GTX 1080ti	164.90	87.95	403.58

Figure 36: cuDNN inference times with varying power modes in half-precision



	VGG 16	Resnet 50	YoloV3
Jetson TX2 (Max-N)	69.72	37.91	172.42
Jetson TX2 (Max-Q)	100.42	52.65	245.45
Jetson TX2 (Max-P Core All)	79.26	42.85	194.66
GTX 1080ti	4.45	2.98	11.67

Figure 37: cuDNN inference times with varying power modes with single-precision

8

CONCLUSION

This dissertation aimed at determining the suitability of the NVidia Jetson TX2 to be the main computing device behind the vehicle cabin sensing system currently in development at Bosch. It also intended to explore its most relevant performance features and determine the most efficient strategy to deploy a neural network based image analysis system on the device.

The embedded device was thoroughly tested and compared against traditional computing hardware to determine its relative performance. For an embedded device of its size, the NVidia Jetson TX2 revealed to be very computationally capable, especially in half-precision floating point operations, making it ideal for deep learning based deployment environments. Another important characteristic is its form factor. The device itself does not contain any I/O, cooling or power delivery system, allowing the system integrator to adapt it to different environments with unique constraints. It can also run in different power consumption modes, further increasing its adaptability to possible heat dissipation of power delivery restrictions. Its main disadvantages is the fact that it does no support OpenCL, forcing any software developed that takes advantage of the GPU to be NVidia dependent, complicating possible hardware platform switches in the future. Another weak point of the device is its very incapable CPU, making it more suited to administrative tasks. Overall, the device suits the needs of the main project as it enables the possibility of using deep learning models to analyse images in real time (as low as 74,66ms per image on YoloV3 model).

This work also tested multiple neural network inference systems to determine the best deployment strategy to be used in the final product. Among the tested strategies is Deeploy, the main implementation effort of this dissertation. This prototype is a neural network inference only system that was specifically designed for this project. It served the purpose of testing two deep neural network libraries developed by NVidia (cuDNN and TensorRT) and, in its current state, can be considered a prototype of a possible deployment solution. This tool development phase accounted for fine-tuning, profiling and validation stages which resulted in an efficient, stable and tested prototype.

After extensive testing, it was possible to conclude that TensorRT is the fastest inference library, by a great margin. This is possible due to the work NVidia employed in making TensorRT efficient and also due to its focus in inference only. It was also concluded that it is possible to fully take advantage of consumer grade hardware to perform neural network training tasks in single floating point precision, with final deployment occurring at a lower precision. The performed tests showed that the reduction in the detection capacity was negligible, with great inference performance gains.

The possibility of exploring the shared memory architecture between the GPU and CPU of the embedded device was also tested. This alternative yielded poor performance, with explicitly memory management still being the fastest alternative.

This work also displayed the importance of fine-tuning and profiling. In the case of the cuDNN based engine, great performance gains were achieved by manually picking the convolution forward algorithm, disregarding the one automatically chosen. In the case of TensorRT, performance gains were achieved by avoiding the use of the plugin layer, since a very large amount of floating point conversions was performed during half-precision inference.

Finally, it it important to mention that, due to its modular approach, Deeploy can easily be extended to test new inference libraries, parse other neural network serialisation formats or implement new neural network layers.

8.1 Future work

The deployment of neural network architectures in embedded systems is a very diverse and competitive space in both software and hardware. Over the course of this dissertation, a prototype inference engine based on two deep neural network libraries was developed. This starting point can be extended to test other libraries, network architectures or hardware systems.

Restricting the project to the same embedded device and libraries, some of the improvements/tests that this dissertation motivates are:

- to explore, in more detail, the embedded device power modes (namely by measuring actual power consumption);
- to implement new neural network architectures to expand on the testing already performed and improve the prototype capabilities;
- to explore the caffe2 framework, due to its claimed high performance in embedded devices;
- to explore the automated caffe2 model import mechanism that TensorRT implements;

- to create a Python library that integrates with the Deeploy to facilitate its use;
- to develop unit tests to automatically validate the project after tuning or other adjustments;
- to assess the possibility of neural network training on the device (e.g.: for automated continuous improvements);
- to assess the inference capabilities of the embedded device CPU;
- to incorporate dynamic power mode selection into the developed tool.

BIBLIOGRAPHY

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI)*, volume 16, pages 265–283, 2016.

Paul Baltusis. On board vehicle diagnostics. Technical report, SAE Technical Paper, 2004.

Christopher M. Bishop. *Pattern recognition and machine learning*. Information science and statistics. Springer, New York, 2006. ISBN 978-0-387-31073-2.

Meredith Broussard. The Dirty Truth Coming for Self-Driving Cars, May 2018. URL https://slate.com/technology/2018/05/who-will-clean-self-driving-cars.html.

Greg Brown. The 2018 uber lost & found index, March 2018. URL https://www.uber.com/newsroom/2018-uber-lost-found-index/.

Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. *arXiv preprint arXiv:1611.08050*, 2016.

Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rudiger Wirth. Crisp-dm 1.0 step-by-step data mining guide. 2000.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *The IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009.

Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

Dustin Franklin. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge, March 2017a. URL https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/.

Dustin Franklin. Nvidia jetson tx2 delivers twice the intelligence to the edge, March 2017b. URL https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/.

Paul Gao, Russel Hensley, and Andreas Zielke. A road map to the future for the auto industry. *McKinsey Quarterly, Oct*, 2014.

Dan Gillick, Cliff Brunk, Oriol Vinyals, and Amarnag Subramanya. Multilingual language processing from bytes. *CoRR*, abs/1512.00103, 2015. URL http://arxiv.org/abs/1512.00103.

Ross Girshick. Fast r-cnn. In *The IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2014a.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *The IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014b.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

Mark Harris. Inside Pascal: NVIDIA's Newest Computing Platform, April 2016. URL https://devblogs.nvidia.com/inside-pascal/.

Mark Harris. Unified Memory for CUDA Beginners, June 2017. URL https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

Nhut-Minh Ho and Weng-Fai Wong. Exploiting half precision arithmetic in nvidia gpus. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016. URL http://arxiv.org/abs/1602.02410.

Adrian Kaehler and Gary Bradski. *Learning OpenCV 3*. O'Reilly Media, Inc., 1st edition, 2017.

Sinan Kaplan, Mehmet Amac Guvensan, Ali Gokhan Yavuz, and Yasin Karalurt. Driver behavior analysis for safe driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):3017–3032, 2015.

Ayoosh Kathuria. What's new in YOLO v3?, April 2018. URL https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Fred Lambert. Majority of 'Tesla Network' revenue will go to owners: 'it's not Tesla versus Uber, it's the people versus Uber', says Musk, October 2016. URL https://electrek.co/2016/10/26/tesla-network-revenue-to-owners-tesla-versus-uber-people/.

Fred Lambert. Tesla Model 3 is equipped with a driver-facing camera for Autopilot and Tesla Network, August 2017. URL https://electrek.co/2017/08/01/tesla-model-3-driver-facing-camera-autopilot-tesla-network/.

Michael Lamm. Dashboards from dial to digital. *Popular Mechanics*, page 150, 1 1984.

Michael Larabel. NVIDIA Rolls Out Tegra X2 GPU Support In Nouveau - Phoronix, 2017. URL https://www.phoronix.com/scan.php?page=news_item&px=Tegra-X2-Nouveau-Support.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

leonardblier. A brief report of the Heuritech Deep Learning Meetup #5, February 2016. URL https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/.

Fei-Fei Li and Andrej Karpathy. Convolutional neural networks for visual recognition. http://cs231n.github.io/, 2015. Accessed: 2018-01-04.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. COCO - common objects in context. http://cocodataset.org/#detection-eval, 2014a.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014b.

Tsung-Yi Lin, Priyal Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

Todd Litman. *Autonomous vehicle implementation predictions*. Victoria Transport Policy Institute Victoria, Canada, 2017.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

Ben Lorica. Why AI and machine learning researchers are beginning to embrace PyTorch - O'Reilly Media, 2017. URL https://www.oreilly.com/ideas/why-ai-and-machine-learning-researchers-are-beginning-to-embrace-pytorch.

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *CoRR*, abs/1803.04014, 2018. URL http://arxiv.org/abs/1803.04014.

Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

NVidia. NVIDIA TensorRT, April 2016. URL https://developer.nvidia.com/tensorrt.

CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL http://arxiv.org/abs/1804.02767.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *IEEE conference on computer vision and pattern recognition (CVPR)*, pages 779–788, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

Ryan Smith. The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation, 2016. URL https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review.

Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Nathan Whitehead and Alex Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. page 7, 2011.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Zhang Xianyi, Wang Qian, and Zaheer Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.