ADAPTING DEEP NEURAL NETWORKS TO A LOW-POWER ENVIRONMENT

Oscar Mañas Sanchez

Director: Antonio González
Co-Director: Jose Maria Arnau
Specialization: Computer Science

Facultat d’Informàtica de Barcelona
Universitat Politècnica de Catalunya
2017
Abstract

These days, working with deep neural networks goes hand in hand with the use of GPUs. Once a deep neural network has been trained for hours, days, or even weeks on a desktop GPU, it is deployed in the field where it runs inference computations, which are far less expensive than training. This fact, in conjunction with the mobile nature of deep learning applications, makes very interesting the possibility of running inference locally on a mobile device.

There already exist mobile applications that can perform tasks involving deep neural networks, but they rely on remote servers to run the most expensive computations. This is not ideal because the user's privacy may be compromised or the algorithm performance may be damaged due to latency issues on a poor network connection. In this project, the possibility of running inference natively on a mobile GPU is explored.

One of the main applications of deep learning is object recognition, which encompasses different problems such as classification, identification and detection. In this project, we select a very deep neural network called Faster R-CNN, which solves a detection problem, and optimize it to run natively on a mobile platform. This innovation provides to a mobile device, such as a smartphone or a tablet, the capability of identifying objects and its location in images, potentially improving the performance of applications that currently employ the device camera combined with deep neural networks.

However, mobile devices have limitations in power, memory, and compute capability. This makes power and memory hungry applications such as deep neural networks hard to deploy, requiring smart software design. As a result, mobile presents both an opportunity and challenge for machine learning systems.

A preliminary profiling of the network on the Nvidia Jetson TX1 module, a state-of-the-art platform used in modern smartphones and handheld consoles, shows that the convolutional and fully-connected layers take most of the forward pass execution time, up to 88.16% of the total. The network parameters take 548.3 MBytes of space, an 87.2% of which belong to fully-connected layers. Hence, the main performance and energy bottlenecks are on the convolutional and fully-connected layers.

In order to overcome these bottlenecks, two optimizations are proposed. In first place, we use half-precision floating points instead of single-precision; this reduces by half the memory bandwidth, improving performance and providing energy savings. In second place, we implement a neuron-pruning technique to remove up to 80% of the neurons in the fully-connected layers; pruning reduces the memory footprint of the network model and the amount of FP operations, reducing both energy consumption and execution time.

3
To evaluate the aforementioned optimizations, thorough experimentation is carried out on a Nvidia Jetson TX1 module. Results show that, combining all the optimizations, we obtain, on average, a speedup of 1.55x, an energy reduction of 31.3%, an improvement in energy-delay of 2.26x and a memory footprint reduction of 86%.
Acknowledgements

My thanks and appreciation to Jose Maria Arnau and Antonio González for persevering with me as my advisors throughout the time it took me to complete this research and write the thesis, generously giving their time and expertise to better my work.

I would also like to thank my loving parents, whose words of encouragement and push for tenacity ring in my ears. Thanks to my girlfriend Maria, who has consistently helped me keep perspective on what is important in life and shown me how to deal with reality.

Finally, I must acknowledge as well the many friends and colleagues who have supported me throughout this process.
Declaration

I declare that this thesis was composed by myself, that the work contained therein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Oscar Mañas Sanchez)
Contents

Abstract 4

Contents 10

List of figures 12

List of tables 13

1 Context and scope of the project 14
   1.1 Context and problem formulation 14
   1.2 Stakeholders 16
      1.2.1 Developer 16
      1.2.2 Director and co-director 16
      1.2.3 Beneficiaries 16
   1.3 State-of-the-art 16
   1.4 Scope 17
   1.5 Methodology 18
      1.5.1 Short development cycle 18
      1.5.2 Intensive client feedback 18
   1.6 Monitoring tools 18
   1.7 Validation methods 18
   1.8 Possible obstacles and solutions 19
      1.8.1 Optimizations selection 19
      1.8.2 Scheduling 19
      1.8.3 Bugs 19
      1.8.4 Computational power 19

CONTENTS 7
2 Project planning

2.1 Planning and scheduling ........................................... 20

2.2 Task description .................................................... 20

2.2.1 Acquire background in deep neural networks ................. 20

2.2.2 Get familiar with Caffe ......................................... 20

2.2.3 Jetson TX1 set up .............................................. 21

2.2.4 Initial neural net characterization ............................. 21

2.2.5 Use half-precision floating points ............................... 21

2.2.6 Pruning of the neural net and retraining ...................... 22

2.2.7 Use cuSPARSE .................................................. 22

2.2.8 Final neural net characterization ............................... 22

2.3 Estimated time ..................................................... 23

2.4 Gantt chart ........................................................ 23

2.5 Alternatives and action plan ...................................... 24

2.5.1 Complexity of the platform ................................... 24

2.5.2 Bugs ............................................................ 24

2.5.3 Optimizations selection ........................................ 24

2.5.4 Unavailability of the Jetson TX1 developer kit .......... 24

2.6 Changes from the initial planning ................................ 25

3 Budget and sustainability ........................................... 27

3.1 Project budget ...................................................... 27

3.1.1 Hardware budget ............................................. 27

3.1.2 Software budget .............................................. 27

3.1.3 Human resources budget ..................................... 28

3.1.4 Unexpected costs .............................................. 29

3.1.5 Indirect costs .................................................. 29

3.1.6 Total budget ................................................... 29

3.2 Budget monitoring ................................................ 30

3.3 Sustainability and social commitment .......................... 30

3.3.1 Environmental dimension ..................................... 31

3.3.2 Economical dimension ....................................... 32

3.3.3 Social dimension ............................................. 32
# List of Figures

2.1 Gantt chart of the project. .................................................. 23  
2.2 Final Gantt chart of the project. ................................. 26  
4.1 Four-layer network with two hidden layers. ......................... 33  
4.2 Deep learning training compared to inference. .................... 35  
4.3 Faster R-CNN network. .................................................... 40  
4.4 Fast R-CNN architecture. ............................................... 41  
4.5 Matrix-matrix multiplication. ........................................... 43  
4.6 Matrix-vector multiplication. ........................................... 43  
4.7 Average execution time per layer. .................................... 45  
4.8 Parameter size per layer. ................................................ 46  
4.9 IEEE 754 half-precision. ................................................. 47  
4.10 Mask layer. .............................................................. 56  
4.11 Loss descent during training. ......................................... 61  
4.12 Execution time applying FP16. ...................................... 63  
4.13 16-bit half-precision data formats. ............................... 64  
4.14 Speedup applying FP16. ............................................... 65  
4.15 Normalized SoC energy applying FP16. ............................. 65  
4.16 Improvement in energy-delay applying FP16. ..................... 66  
4.17 Execution time applying neuron-pruning .......................... 68  
4.18 Speedup applying neuron-pruning ..................................... 69  
4.19 Normalized SoC energy applying neuron-pruning .................. 70  
4.20 Improvement in energy-delay applying neuron-pruning .......... 71  
4.21 Average precision per class applying neuron-pruning .......... 72
4.22 Execution time applying FP16 and neuron-pruning. . . . . . . . . . . . . 73
4.23 Speedup applying FP16 and neuron-pruning. . . . . . . . . . . . . . . . 74
4.24 Normalized SoC energy applying FP16 and neuron-pruning. . . . . . . 75
4.25 Improvement in energy-delay applying FP16 and neuron-pruning. . . . 76
4.26 Memory footprint of the different optimizations. . . . . . . . . . . . . . . 77
List of Tables

2.1 Estimated time for each task ........................................... 23
2.2 Time spent on each task .................................................. 26

3.1 Hardware budget .......................................................... 27
3.2 Software budget .......................................................... 28
3.3 Human resources budget .................................................. 28
3.4 Time estimation by role .................................................... 29
3.5 Unexpected costs .......................................................... 29
3.6 Indirect costs ............................................................... 29
3.7 Total budget ............................................................... 30
3.8 Total budget ............................................................... 30
3.9 Sustainability matrix of the project .................................... 31

4.1 Shape per blob ............................................................. 44

B.1 Execution time per layer .................................................. 90

C.1 Parameter size per layer ................................................... 91
Chapter 1

Context and scope of the project

1.1 Context and problem formulation

The holy grail of computing is artificial intelligence: building a machine so intelligent, it can learn on its own without explicit instruction. Machine-Learning algorithms have become widespread in a very broad range of applications and cloud services. These applications are displacing scientific computing as the major driver for high-performance computing. While this profound shift in applications is occurring, the machine-learning domain has greatly evolved since 2006, where a category of algorithms, called deep learning (convolutional and deep neural networks), has emerged as state-of-the-art across a broad range of applications [1, 2, 3, 4, 5].

Deep learning is a critical ingredient for achieving modern AI, and it is revolutionizing the world. In the last years, deep neural networks (DNN) have been conquering, one by one, various algorithm domains related to computer vision in particular and machine perception in general. There is an endless number of potential use cases: from self-driving cars [6] to pharmaceutical research [7] and faster drug development [8], from automatic image captioning [9] in online image databases to smart real-time language translation [10] in video chat applications. Even in the field of art, there have been developed applications for transferring the pictorial style of a painting onto an image [11].

Deep learning allows the AI brain to perceive the world around it; the machine learns and ultimately makes decisions by itself. It takes massive amounts of data to train the machine to do this. In addition, highly sophisticated deep neural networks are needed to process it all. In 2012, Google’s Deep Learning project, Google Brain, learned to recognize cats by watching movies on YouTube [4]. But it required 2,000 CPUs (16,000 CPU cores) in servers powered and cooled in one of Google’s data centers to do this. Few organizations have machines of this scale. Around the same time, NVIDIA Research teamed with Stanford University to use GPUs for deep learning. It turned out that 12 GPUs could deliver the deep learning performance of 2,000 CPUs.

That is the reason why these days, working with deep neural networks goes hand in hand with the use of GPUs. On a high level, working with deep neural networks is a two-stage process: First, a neural network is trained, i.e. its parameters are determined using labeled examples of inputs and desired output. Once a deep neural network has
been trained for hours, days, or weeks, the next step is to put it to use by deploying it in the field, where it will repeatedly run inference computations. Examples of inference (giving an input to a DNN which then extracts information based on that input) include classifying images, localizing faces, or translating human speech in real-time.

Training and inference have different performance goals. Training strives for throughput (because it is common to batch hundreds of training inputs), whereas inference strives for latency (because users do not want to wait several seconds for their input to be processed). Furthermore, the computational requirements for running inference are a lot less than for training a deep neural network. This, in conjunction with the mobile nature of some of the before mentioned applications, make the next logical step pretty clear: be able to run inference live, in real time, on people’s devices (i.e. run inference in a low-power environment using a mobile GPU), while the training step is done beforehand using dedicated hardware (i.e. high end GPUs).

There already exist mobile applications that are capable of perform this kind of tasks [12], but they rely on the cloud to run the most computationally expensive parts. This is not ideal for several reasons. First, it does not guarantee the user’s privacy because the data is being sent to remote servers. Second, the algorithm performance may be damaged due to latency issues on a poor network connection. In addition, sending continuous data to a server may be too expensive. In this context, this project aims to make mobile devices perceive, process and comprehend in real time, without having to connect to remote servers.

Nevertheless, running large neural networks requires a lot of memory bandwidth to fetch the weights and a high computational power to perform a lot of floating point operations, and mobile devices have limitations. While they have improved significantly in computation power in the last years and are capable of carrying out billions of arithmetic computations per second, they also have various resource constraints like power, memory, and compute capability. This makes power and memory hungry applications such as deep neural networks hard to deploy, requiring smart software design. As a result, mobile presents both an opportunity and challenge for machine learning systems.

The objective of this project is to be able to run in real time a specific application like FASTER R-CNN [13] that does inference using a particular deep learning framework like CAFFE [14]. The application takes an image as input and outputs a list of regions with predictions of objects inside those regions. It uses a DNN that combines a Region Proposal Network (RPN) and a Fast Region-based Convolutional Network (Fast R-CNN); the RPN component tells the unified network where to look, and then the Fast R-CNN component is used to perform object detection inside the regions of interest. The application will be run in a concrete low-power environment like the NVIDIA JETSON TX1 [15], which is a high-end mobile platform with a low-power GPU (NVIDIA Tegra X1) used in commercial products. To accomplish this, several software optimizations will be applied.
1.2 Stakeholders

1.2.1 Developer

Is the person in charge of research, document and implement all the required software. In addition, he/she is responsible for the project management and the writing of the report and all the required documentation. This actor works as agreed with the director and co-director and he is, ultimately, the person in charge of accomplish the deadlines.

1.2.2 Director and co-director

They are the main responsible for guiding, giving advice and, in general, helping the developer. Their action is key to determine possible errors in the project, both in its proposal and execution. In particular, Antonio González from the Computer Architecture Department (DAC) and leader of the ARCO research group, has acted as director. Jose Maria Arnau, also from the DAC, has acted as co-director.

1.2.3 Beneficiaries

Since this project is not about creating a product, it may seem that it does not have any direct beneficiary. Nevertheless, optimize and adapt an application that uses deep neural networks to a low-power environment may bring knowledge to researches interested in similar topics. In addition, smartphone users can benefit from these optimizations and have a better user experience when using applications that implement them. Furthermore, this project can also serve as a starting point for future projects that also use the Jetson TX1 platform.

1.3 State-of-the-art

This is a very hot topic, so there is a lot of research on this subject going on these days. Some researchers are trying to minimize the memory footprint using lower precision and approximate computing, and reducing the cost of convolutions with FTT [16]. Others are trying to reduce the size of deep neural networks by factoring the learned parameters to keep them on a constrained rank [17].

Another approach is CNN compression. In [18], a three stage pipeline is aimed to reduce the storage requirement of neural networks. The network is first pruned by learning only the important connections. Next, weights are quantized to enforce weight sharing. Finally, Huffman coding is applied to the weights. In [19], the process also consists in three steps. The first step is rank selection with variational Bayesian matrix factorization. Second, Tucker decomposition is applied on kernel tensors. Finally, the network is fine-tuned to recover the accumulated loss of accuracy.

Other applications allow to further optimize the model. One example is the spotting of audio keywords like “Ok, Google”, which uses a finished trained model on the device [20].
Another improvement is the final fine-tuning of the model and adaption of some parameters on the target device \cite{21}. For example, the model is pre-trained to recognize faces. Then, on the mobile device, the network is trained to recognize one specific face \cite{22}.

One example of a real application running deep learning algorithms inside a phone is Google Translate \cite{23}. To achieve real-time, the Google team heavily optimized and hand-tuned the math operations, using the mobile processor’s SIMD instructions and tuning things like matrix multiplies to fit processing into all levels of cache memory.

Maybe the most similar approach to what is being proposed in this project is the work of folks at Facebook with Caffe2Go \cite{24}. They designed a lightweight and modular framework by building on top of the open-source Caffe2 project \cite{25}. Because of the modular design, the framework can speak the same language but be optimized for each platform. When the graph is actually run, it instantiates itself with the various hardware features to achieve maximum speed.

1.4 Scope

The first task will be getting up and running the application Faster R-CNN in the Jetson TX1 platform. Next, a profiling of the application will be performed in order to identify the main bottlenecks and determine the parts where the optimizations are most needed.

Since the application is built with Python on top of the Caffe deep learning framework, which is written in C++, the improvements will be focused on Caffe. The optimizations will be centered in reducing the memory bandwidth and the energy required to run inference in the Jetson TX1 platform, which in turn will contribute to reduce the inference time. In that sense, the long term goal is to achieve real-time processing speed. That means about 20-30 frames per second, i.e. about 30-50 ms to process a single image. But this is a pretty ambitious goal and is unlikely to be reached, since the current time per frame in the Jetson TX1 platform is over 2 seconds. The actual goal will be to get as close as possible to a real-time value.

The list of optimizations will include exclusively software optimizations, like using half-precision floating point data types, reusing the same memory buffer between layers, using data compression algorithms to minimize the size of the weights, pruning connections with weights close to zero and applying a retraining step afterwards, and using cuSPARSE\cite{26} for handling the sparse matrices resulting from the pruning. The use of half-precision floats is especially attractive, not only because it contributes to reduce the memory footprint, but also because mobile platforms offer FP16 units not available in desktop, which double the throughput with respect to FP32 units according to Nvidia. As a disclaimer, since the project is still at a pretty early stage, these are only some ideas of the optimizations that may be performed. As the project develops, some of these may be discarded and some others may be added.

After each optimization, the code will be tested in terms of accuracy and efficiency. Regarding the accuracy, a series of unit tests will be run to ensure that the application is still giving a correct result. To test efficiency, the code will be instrumented with several timers in order to measure how much time is spent in each of the stages of the algorithm.
1.5 Methodology

As a result of the tight timetable the project will be developed in, an agile approach seems the best fit. Agile methodologies provide flexibility, fast development and results in less time than any other ordinary methodologies. However, currently accepted agile methodologies are very team-oriented, so strictly following any of them would make little sense. Nonetheless, some of these methodologies concepts are still valid for solo projects.

1.5.1 Short development cycle

By using an iterative approach with short cycles (goals will be specified on a weekly basis), it is much easier to keep the project on schedule and be conscious about the current state of the project. On a higher level, iterations will be done on a feature basis. That is, each iteration will focus on an optimization and will consist of the following stages: (1) profile of the application to detect bottlenecks, (2) think about the best way to optimize the code, (3) implement the optimization and (4) evaluate if it has been enough improvement. If the answer is no, the next iteration will focus again on the same optimization.

1.5.2 Intensive client feedback

Although the project does not have a real client, this concept from agile methodologies still applies: by letting the person that is ultimately responsible for evaluating the project check the project state as frequent as possible, chances for misunderstandings are greatly reduced and, if they occur, are corrected much faster.

1.6 Monitoring tools

Git [27] and GitHub [28] are going to be the tools used to monitor the evolution of the project. Git enforces a short cycle approach (by means of commits) and forces the developer to document all changes. GitHub provides a remote Git repository and an integrated issue tracking system, perfect for setting milestones and tracking the project’s progress.

1.7 Validation methods

The combination of unit tests, GitHub issues and weekly meetings with the project tutor ensure the validation of the objectives on their own without need for further measures.
1.8 Possible obstacles and solutions

1.8.1 Optimizations selection

A bad selection of the optimizations to be performed could result in a waste of time without a significant gain. That is why the selection of the optimizations will be guided by Amdahl’s law: the first optimizations to be implemented will be those focusing on parts that have a greater margin of improvement. This way, the time spent in each optimization will be proportional to the theoretical gain that could be obtained if it is correctly implemented.

1.8.2 Scheduling

Four months can seem like a long time, but a bad scheduling can make it insufficient. To avoid this problem, a very rigid but realistic timetable with weekly meetings with the project tutor will be set to ensure the project is on schedule, or to apply the appropriate corrections otherwise.

1.8.3 Bugs

Considering that a deep learning framework is a relatively complex software, it’s easy to introduce bugs while modifying the source code. To ensure no bugs are present, a series of unit tests will be run after each optimization to verify that the application is still giving a correct result.

1.8.4 Computational power

Some of the optimizations may require a retraining step. While running inference is pretty cheap in terms of computational power, training needs a lot more power since hundreds of millions of inputs are passed forward and backward through the network. Shall this be the case, using the ARCO research group hardware would probably be enough.
Chapter 2

Project planning

2.1 Planning and scheduling

The estimated project duration is of about 4 months. The project starts on Wednesday 1st February, 2017 and the deadline is on Friday 23rd June, 2017, the day before presentations start.

As a disclaimer, it must be pointed out that the initial planning could be revised and updated as a result of the evolution of the project. Furthermore, the use of agile methodologies implies that new requirements that alter the original planning may appear.

2.2 Task description

2.2.1 Acquire background in deep neural networks

In the last months I have been learning about deep learning in general and deep neural networks in particular. I started by reading an online book called *Neural Networks and Deep Learning* [29] that gave me a general view on how artificial neural networks work, from simple perceptrons to more complex convolutional neural networks, and how they are trained. From there, I took a course on Coursera titled *Neural Networks for Machine Learning* [30] imparted by Geoffrey Hinton (University of Toronto). Once I got familiar with all the concepts related to neural networks, I read three articles involved in the particular application that is going to be examined in this project [31, 32, 33]. This process took several months because I was also studying my seventh full time semester at FIB. This task did not require any material resources (except for the printed documents), but it did require human resources to read and understand all the information.

2.2.2 Get familiar with Caffe

A deep learning framework is a pretty complex piece of software, and the kind of optimizations we will be doing require a deep understanding of the intricacies of the source
code of Caffe: how data is organized, how memory is managed, etc. That is why, before starting to optimize this framework, a deep dive into the source code of Caffe is required. This task will take several weeks, but it can be done in parallel with the GEP course. As a material resource, the Caffe deep learning software is required. This task also requires human resources to read and understand the source code.

2.2.3 Jetson TX1 set up

The whole project will be developed in a specific high-end mobile platform: the NVIDIA Jetson TX1, which packs a Tegra X1 GPU and an ARM Cortex-A57 processor. Despite being a mobile platform, it can run a Desktop OS like Ubuntu. The first step is installing Ubuntu 14.04 in the system and, after that, installing the Caffe dependencies. Luckily enough, NVIDIA also ships an all-in-one package called JetPack which bundles and installs all software tools required to develop for the platform, such as cuDNN and CUDA. The last step is forking the Caffe repository from GitHub, build it and run some tests in order to make sure that everything works properly. Once the Jetson TX1 developer kit arrives, this task should not take longer than a day. As a material resource, this task will require the Jetson TX1 developer kit and all the software necessary to run Caffe. This task also needs human resources to set up the Jetson TX1 module and install the required software.

2.2.4 Initial neural net characterization

Before applying any optimization, first we have to find out where the bottlenecks are, i.e. which parts of the code up the bulk and execution time. This is called profiling or performance characterization: analyze the program to obtain detailed information about execution time, memory usage, energy consumption, etc. A Python script will be written in order to obtain information about execution time and memory. As for energy consumption, the Jetson TX1 developer kit has hardware counters that allow to extract how much energy is consuming a particular running application. The idea is to extensively profile the application at first, and then a less detailed profiling will be run after each optimization. This task should take several days, and it requires the Jetson TX1 developer kit and all the involved software as a material resource. This task also needs human resources to write the scripts that will be used to collect the profiling data and then analyze that data to detect bottlenecks.

2.2.5 Use half-precision floating points

This optimization consists in substituting all the single-precision floats used in the application for half-precision floats. The motivation behind this optimization is that regular, single-precision floats take 32 bits of memory, but half-precision floats take only 16 bits. This not only will cut the memory footprint by half, but will allow us to benefit from FP16 units only present in mobile GPUs like the Tegra X1: NVIDIA claims that FP16 units can deliver two times more throughput than regular FP32 units, so the execution time will also be reduced. Furthermore, this optimization is extensively supported by the literature: some studies have shown that the use of half-precision floats incurs little to no degradation in the classification accuracy [34]. This task should
take several weeks, depending on the easiness of adapting the whole application to half-precision floats. As material resources, it requires the Jetson TX1 developer kit and all the involved software. This task also needs human resources to adapt the application to FP16.

2.2.6 Pruning of the neural net and retraining

The next optimization consists in pruning the DNN to reduce memory footprint and computation time. Once the neural net has been trained, each connection has a weight assigned. These weights are then used in the forward pass to perform the operations that ultimately allow to classify an image. But if a weight is close to 0, it has little to no impact in the final decision of the net. That is why connections with weights close to zero can be pruned away with little impact on accuracy. More aggressive pruning can be employed to further reduce the size of the DNN. However, very aggressive pruning (around 90% of the net) has a significant impact on accuracy. In that case, weights which have not been pruned need to be adjusted. Hence, a retraining step is needed after pruning. Since the retraining of the net alone could take up to a week, this task may take several weeks. As a material resource, this task requires the Jetson TX1 developer kit and all involved software. This task also needs human resources to write the scripts needed to do the pruning and to perform the retraining of the net.

2.2.7 Use cuSPARSE

The last optimization is a logical step after performing the pruning. In Caffe, weights are represented as multi-dimensional matrices, so it is not surprising that a great portion of the computation consists in operations with matrices. The previous optimization has the side effect of converting most of these matrices in sparse matrices since pruned weights are set to zero, and operations with sparse matrices can be exploited to improve performance over dense linear algebra. By default, Caffe uses cuBLAS to perform matrix operations. But if these matrices are sparse, there is a better solution: cuSPARSE. Therefore, the optimization consists in substituting every call to a cuBLAS routine for a call to a cuSPARSE routine. This task should not take more than a few days, and it requires the Jetson TX1 developer kit and all the involved software (particularly cuSPARSE) as a material resource. This task also needs human resources to adapt the application to cuSPARSE.

2.2.8 Final neural net characterization

The final task consists in reporting the gains obtained with the aforementioned optimizations. To this extent, the application will be characterized again measuring execution time, memory usage, energy consumption, etc. This task should take several days, and it requires the Jetson TX1 developer kit and all the involved software as a material resource. This task also needs human resources to run the scripts that will be used to collect the profiling data and then analyze that data to detect improvements with respect the initial version of the application.
2.3 Estimated time

In table 2.1, an estimation of the number of hours dedicated to each task is shown.

<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated duration (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire background in DNN</td>
<td>70</td>
</tr>
<tr>
<td>Get familiar with Caffe</td>
<td>40</td>
</tr>
<tr>
<td>Jetson TX1 set up</td>
<td>10</td>
</tr>
<tr>
<td>Initial net characterization</td>
<td>30</td>
</tr>
<tr>
<td>Use of half-precision floats</td>
<td>90</td>
</tr>
<tr>
<td>Pruning and retraining</td>
<td>90</td>
</tr>
<tr>
<td>Use of cuSPARSE</td>
<td>60</td>
</tr>
<tr>
<td>Final net characterization</td>
<td>30</td>
</tr>
<tr>
<td>Final stage</td>
<td>30</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>450</strong></td>
</tr>
</tbody>
</table>

*Table 2.1: Estimated time for each task.*

2.4 Gantt chart

Figure 2.1 shows the planning of the different tasks of the project in a Gantt chart.

*Figure 2.1: Gantt chart of the project. Generated with https://www.teamgantt.com*
2.5 Alternatives and action plan

As it was pointed out in the previous delivery, the agile methodology will allow to revise and adapt dynamically this initial planning. Therefore, if the real duration of the specified tasks differs from the expected duration, the planning will be modified accordingly. In the (rare) case that a task has less duration that expected, the next task will immediately start. On the other hand, if a task lasts more than expected, a later task will have to be shortened or, in the worst case, completely omitted. The mentioned optimizations are decreasingly ordered based on the estimated gain in terms of execution time, memory usage and energy consumption. So if the project was running short on time, the last optimization could be cut out.

Since weekly meetings have been arranged with the project tutor, there will be enough time to detect possible deviations from the original planning and correct them.

In conclusion, with an estimated dedication of 30 hours per week and a total budget of (16x30) 480 hours, the project planning is feasible.

Next, some examples of potential sources of delays are mentioned.

2.5.1 Complexity of the platform

The Caffe deep learning framework is not a final product, and it is constantly being developed by the community. The lack of documentation and information may generate some problems or difficulties, which in turn may cause delays in the development of the optimizations.

2.5.2 Bugs

Considering that a deep learning framework is a relatively complex software, it is easy to introduce bugs while modifying the source code. This may also cause delays if bugs are not located and fixed fast enough.

2.5.3 Optimizations selection

A bad selection of the optimizations to be performed could result in a waste of time without a significant gain. This could cause delays with respect to the planning and carries the risk of not being able to finish the implementation of all the proposed optimizations.

2.5.4 Unavailability of the Jetson TX1 developer kit

It may happen that due to reasons beyond our control, we are unable to access remotely the machine, like for example a disconnection from the network, problems with credentials, etc. If the unavailability of the machine persists long enough, it could cause deviations in the project planning.
2.6 Changes from the initial planning

During the development of this project, there have been some deviations with respect to the initial planning. In this section we explain the reasons of these deviations and how we solved them.

Originally, the plan was to prune the deep neural network using a technique called *Deep Compression* [18]. This technique prunes weights and produces sparse matrices, so a subsequent adaptation of the Caffe source code would have been necessary to employ sparse instead of dense linear algebra. More specifically, all the calls to the cuBLAS library should have been replaced by calls to the cuSPARSE library. This comprises the last two implementation tasks of the original planning: weight-pruning and using cuSPARSE. Due to the magnitude of the modifications in the Caffe framework, the previous task, consisting in using half-precision floats, lasted some more weeks than expected.

In order to finish the project on time, we decided to change the pruning technique. Instead of pruning weights, we decided to go with a neuron-pruning technique. The advantage of neuron-pruning is that the resulting matrices (see section 4.2.1.1) are smaller but not sparse, so the Caffe source code does not need to be modified to change the linear algebra library from cuBLAS to cuSPARSE. Nevertheless, neuron-pruning offers potentially smaller benefits due to pruning at bigger granularity.

Furthermore, the controversy generated around the performance of weight-pruning made us hesitate about this pruning technique. Recently, a paper submitted to the ISCA 2017 [35] claimed that the weight-pruning carried out in [18] hurts performance despite removing an average of 80% of the weights due to matrix sparsity (see section 4.4.1). Given these contradictory results, we believe that focusing on neuron-pruning entails a smaller risk for the project as both memory and performance improvements are granted.

In view of all this, we decided to modify the original planning. The last two optimizations of the project, weight-pruning and using cuSPARSE, were replaced by an optimization consisting in pruning neurons instead of weights (see section 4.4.2.2). This change of plans brought us back on schedule, since this last optimization took a shorter implementation time than the previous two together.

These changes did not affect the original objective of the project. In addition, these changes did not have any impact on costs either, since they did not require additional resources and the implementation time was roughly the same as in the original planning.

Table 2.2 shows an updated version of the number of hours dedicated to each task.
Table 2.2: Time spent on each task.

<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated duration (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire background in DNN</td>
<td>70</td>
</tr>
<tr>
<td>Get familiar with Caffe</td>
<td>40</td>
</tr>
<tr>
<td>Jetson TX1 set up</td>
<td>10</td>
</tr>
<tr>
<td>Initial net characterization</td>
<td>30</td>
</tr>
<tr>
<td>Use of half-precision floats</td>
<td>140</td>
</tr>
<tr>
<td>Neuron-pruning and retraining</td>
<td>100</td>
</tr>
<tr>
<td>Final net characterization</td>
<td>30</td>
</tr>
<tr>
<td>Final stage</td>
<td>30</td>
</tr>
<tr>
<td>Total</td>
<td>450</td>
</tr>
</tbody>
</table>

Figure 2.2 shows the final planning of the different tasks of the project in a Gantt chart.

Figure 2.2: Final Gantt chart of the project. Generated with [https://www.teammant.com](https://www.teammant.com)
Chapter 3

Budget and sustainability

3.1 Project budget

In order to carry out this project, the resources mentioned in the previous delivery will be needed. In this document, an estimation of the cost of the project is presented, taking into account the aforementioned hardware and software resources, and the corresponding amortizations. In addition, the indirect costs of the project are taken into consideration.

3.1.1 Hardware budget

In order to implement the optimizations previously explained, a set of hardware will be needed for different purposes.

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Units</th>
<th>Useful life</th>
<th>Amortization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toshiba Portégé R30-A-17E</td>
<td>900 €</td>
<td>1</td>
<td>5 years</td>
<td>75 €</td>
</tr>
<tr>
<td>Jetson TX1 developer kit</td>
<td>450 €</td>
<td>1</td>
<td>3 years</td>
<td>75 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,350 €</strong></td>
<td><strong>150 €</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Hardware budget

Table 3.1 shows an estimation of the cost of the hardware used in the project, taking into account its useful life as well as its amortizations.

3.1.2 Software budget

Additionally, some software will be needed to carry out the project. Since this project is completely built with free tools, the total software cost is zero. Furthermore, the amortizations are also zero.
Table 3.2: Software budget

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Units</th>
<th>Useful life</th>
<th>Amortization</th>
</tr>
</thead>
<tbody>
<tr>
<td>JetPack (Jetson SDK)</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Caffe</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>LaTeX</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Atom</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Git</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>GitHub</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Evince</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Vim</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>Google Docs</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td>TeamGantt</td>
<td>0 €</td>
<td>1</td>
<td>-</td>
<td>0 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>0 €</td>
<td></td>
<td></td>
<td>0 €</td>
</tr>
</tbody>
</table>

Table 3.2 shows an estimation of the cost of the software used in the project, taking into account its useful life as well as its amortizations.

### 3.1.3 Human resources budget

This project is going to be developed by one person. Hence, this person will need to be a project manager, a software developer and a tester. Thus, each role is differentiated in the total of 450 hours. In table 3.3 an estimation of the human resources cost is provided.

Table 3.3: Human resources budget

<table>
<thead>
<tr>
<th>Role</th>
<th>Hours</th>
<th>€/hour</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>50</td>
<td>50</td>
<td>2.500 €</td>
</tr>
<tr>
<td>Software developer</td>
<td>300</td>
<td>35</td>
<td>10.500 €</td>
</tr>
<tr>
<td>Tester</td>
<td>100</td>
<td>30</td>
<td>3.000 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>450</td>
<td></td>
<td>16.000 €</td>
</tr>
</tbody>
</table>

Next, table 3.4 provides a distribution of the time that each role spends in the different tasks of the project.

CHAPTER 3. BUDGET AND SUSTAINABILITY
### Table 3.4: Time estimation by role

<table>
<thead>
<tr>
<th>Task</th>
<th>Project Manager</th>
<th>Software developer</th>
<th>Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire background in DNN</td>
<td>70</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Get familiar with Caffe</td>
<td>40</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Jetson TX1 set up</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Initial net characterization</td>
<td>30</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Use of half-precision floats</td>
<td>90</td>
<td>0</td>
<td>70</td>
</tr>
<tr>
<td>Pruning and retraining</td>
<td>90</td>
<td>0</td>
<td>70</td>
</tr>
<tr>
<td>Use of cuSPARSE</td>
<td>60</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>Final net characterization</td>
<td>30</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Final stage</td>
<td>30</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>450</strong></td>
<td><strong>50</strong></td>
<td><strong>300</strong></td>
</tr>
</tbody>
</table>

#### 3.1.4 Unexpected costs

Table 3.5 shows the number of extra hours resulting from the worst case of planning deviation. This allows to have a certain budget margin that will help against unexpected events that may occur during the execution of the different tasks.

<table>
<thead>
<tr>
<th>Role</th>
<th>Hours</th>
<th>€/hour</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>15</td>
<td>50</td>
<td>700 €</td>
</tr>
<tr>
<td>Software developer</td>
<td>15</td>
<td>35</td>
<td>525 €</td>
</tr>
<tr>
<td>Tester</td>
<td>10</td>
<td>30</td>
<td>300 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>40</strong></td>
<td><strong>30</strong></td>
<td><strong>1,525 €</strong></td>
</tr>
</tbody>
</table>

#### 3.1.5 Indirect costs

Next, table 3.6 provides an estimation of the indirect costs that are not comprised in any of the previous categories.

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Units</th>
<th>Estimated cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electricity</td>
<td>0.07 €/kWh</td>
<td>1500 kWh</td>
<td>102 €</td>
</tr>
<tr>
<td>ADSL</td>
<td>35 €/month</td>
<td>4 months</td>
<td>140 €</td>
</tr>
<tr>
<td>Office supplies</td>
<td>150 €</td>
<td>-</td>
<td>150 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>392 €</strong></td>
</tr>
</tbody>
</table>

#### 3.1.6 Total budget

By adding all the budgets provided above, the total estimated budget for this project is computed, as shown in table 3.8. Notice that a 5% of contingency has been added.
over the cumulative total in order to cover unexpected expenses that may occur during the course of the project.

Table 3.7: Total budget

<table>
<thead>
<tr>
<th>Concept</th>
<th>Estimated cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware resources</td>
<td>150 €</td>
</tr>
<tr>
<td>Software resources</td>
<td>0 €</td>
</tr>
<tr>
<td>Human resources</td>
<td>16,000 €</td>
</tr>
<tr>
<td>Unexpected costs</td>
<td>1,525 €</td>
</tr>
<tr>
<td>Indirect resources</td>
<td>392 €</td>
</tr>
<tr>
<td><strong>Subtotal</strong></td>
<td><strong>18,067 €</strong></td>
</tr>
<tr>
<td>Contingency (5%)</td>
<td>903.35 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>18,970.35 €</strong></td>
</tr>
</tbody>
</table>

Table 3.8: Total budget

3.2 Budget monitoring

In order to control the budget, at the end of each task the budget will be updated with the effective amount of hours, the cost of the resources used and the expenses of the unexpected events that may have occurred. These numbers will be compared with the previous estimations to obtain indicators that will show the amount of deviation from the initial budget planning. The following formulas will be applied:

\[
\text{Cost deviation} = (EC - RC) \cdot RH
\]
\[
\text{Consumption deviation} = (EH - RH) \cdot EC
\]

where:

- \(EH\) = estimated hours
- \(EC\) = estimated cost
- \(RH\) = real hours
- \(RC\) = real cost

Since the budget will be updated at the end of each task, this will help determine where the deviation has occurred, if it has occurred at all. In addition, since both the amount of hours spent in each task and the cost of the resources used in each task are tracked, this will help determine whether the deviation is due to a variance in cost or consumption.

Furthermore, a certain margin of deviation is allowed since the cost of unexpected events has been taken into account and a contingency percentage has been applied to the final budget estimation.

3.3 Sustainability and social commitment

With the object of identifying and evaluating the sustainability of the project, the impact in its environment will be analyzed regarding three dimensions: environmental,
economical and social. This analysis will be based on the application of the sustainability matrix to the project, shown in table 3.9.

<table>
<thead>
<tr>
<th></th>
<th>PPP</th>
<th>Useful life</th>
<th>Risks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Environmental</strong></td>
<td>Design consumption</td>
<td>Ecological footprint</td>
<td>Environmental risks</td>
</tr>
<tr>
<td></td>
<td>9/10</td>
<td>18/20</td>
<td>-2/-20</td>
</tr>
<tr>
<td><strong>Economical</strong></td>
<td>Bill</td>
<td>Viability  plan</td>
<td>Economical risks</td>
</tr>
<tr>
<td></td>
<td>6/10</td>
<td>15/20</td>
<td>-2/-20</td>
</tr>
<tr>
<td><strong>Social</strong></td>
<td>Personal impact</td>
<td>Social impact</td>
<td>Social risks</td>
</tr>
<tr>
<td></td>
<td>8/10</td>
<td>16/20</td>
<td>0/-20</td>
</tr>
<tr>
<td><strong>Sustainability range</strong></td>
<td>23/30</td>
<td>49/60</td>
<td>-4/-60</td>
</tr>
</tbody>
</table>

Table 3.9: Sustainability matrix of the project

### 3.3.1 Environmental dimension

The energy consumption of the project is relatively small and has little environmental impact, since the majority of the work is done in a low-power laptop. For the same reason, it is difficult that some resources can be reused in other projects, or that problems arise during its dismantling. According to official data [36] and the energy consumption estimated in section 3.1.3, the total amount of emissions produced during the development of this project is estimated in 462 kg of CO$_2$.

Notice that the execution of this project uses the minimum amount of resources possible, limited only to the electricity required for the equipment to work. This fact limits the search of alternatives to reduce the consumption and the environmental impact. This also makes the reuse of resources difficult.

Moreover, this project aims to reduce the energy consumption of inference in deep neural networks. Hence, not only it does not have almost any ecological impact, but also improves the ecological footprint of current solutions, which have the added cost of sending data to remote servers. The proposed solution does all the computation inside the device itself, thus all costs associated to the maintenance of networks, routers, switches and remote servers will be reduced.

A dilation in the project duration would imply an increment in the consumption of resources, which translates into higher CO$_2$ emissions and an increase in the ecological footprint. Therefore, the bigger risk from an environmental point of view is the duration of the development of the project. However, this risk should not represent an important problem since the energy consumption is already very tight, consisting solely in the energy necessary to run the machines used in the development of this project. Furthermore, the massive utilization of the implemented optimizations would represent a notable reduction in the energy consumption of mobile devices that make use of deep neural networks.

Finally, environmentally friendly measures have been taken during the development of this project, such as the use of recycled paper, printing of documents only when strictly necessary and the shutdown of the machines when they are not being used.
3.3.2 Economical dimension

A detailed quantification of all the costs involved in the project has been done, both of material and human resources, as shown in previous sections of this document. These costs correspond with the estimated costs in section 3.1, demonstrating that the economical variability the project can suffer is very low.

Since this is a research and innovation project, no maintenance or additional cost is needed during its lifespan. This contributes to the fact that the final budget is not too high in comparison with other projects that do need maintenance. Moreover, the proposed solution will be less expensive than current solutions from an economical point of view, simply because the ability to run inference directly on the device makes unnecessary a network connection. Hence, this solution will be cheaper for the end-user, since it will not require a data plan to work. This assures the viability of the project in case it had to take its place in a competitive market.

It would be possible to accelerate the development of this project if a more experimented person would have been hired, but this could also increment the cost due to a higher salary. However, at the end of the project it has been proven that the amount of time assigned to the project tasks has been correctly distributed in a proportional way depending on the tasks importance. Although there have been some deviations during the course of this project, they have not affected the project budget.

From an economical point of view, the biggest risk of this project is a possible indisposition of the developer, which would increment the number of hours required to finish the project. This would directly translate into an increment in the cost of the basic resources (electricity, ADSL, etc) and the human resources (salaries). However, the unexpected costs included in the budget would cover a rational temporal deviation, leaving a huge dilation of the development time of the project as the only potential risk.

3.3.3 Social dimension

The execution of this project has not implied significant reflections in a personal, professional or ethic level since, in any case, it would only entail improvements regarding the execution time, memory usage or energy consumption of inference in deep neural networks. This makes unnecessary the consideration of negative scenarios in which some groups would be adversely affected by the project.

As mentioned before, this is currently a hot topic and a lot of research is being carried out on this matter. Hence, collectives of researchers involved in this topic could benefit from this work, but in no case it would be detrimental. In addition, it could improve the quality of life of all users of applications powered by deep neural networks, since these systems could run faster and consume less energy.

The implementation of the technology and methods proposed in this project remains transparent to the users. The population, therefore, would experiment the aforementioned improvements passively, without noticing further change than a better user experience. Hence, this project does not create neither a detrimental scenario nor a dependency that could leave users in a weak position.
Chapter 4

Design and implementation

4.1 Introduction

4.1.1 Deep Neural Networks in a Nutshell

An Artificial Neural Network (ANN) can be seen as a large collection of connected simple units called artificial neurons. A perceptron is a type of artificial neuron, and it can be thought as a mathematical function, which takes several binary inputs (representing dendrites) and produces a single binary output (representing a neuron’s axon). The neuron’s output, 0 or 1, is determined by whether the weighted sum of the inputs is less than or greater than some threshold value. This way, connections between neurons carry an activation signal of varying strength. If the combined incoming signals are strong enough, the neuron becomes activated and the signal travels to other neurons connected to it. Perceptrons can be thought as devices that make decisions by weighting up evidence. By varying the weights and the threshold, we can get different models of decision-making [29].

Figure 4.1: Four-layer network with two hidden layers.
The activation function of a node defines the output of that node given an input or set of inputs. In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing in the neuron. In its simplest form, this function is binary—that is, either the neuron is firing or not. The function described above is of this form, named binary step, but there exist many more, such as logistic, tanH, ReLU, sigmoid... each one with different properties.

Typically, neurons are connected in layers, and signals travel from the first (input), to the last (output) layer. Layers between the input and the output layers are called hidden layers. While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers; neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behavior they want out of their nets.

A Deep Neural Network (DNN) is an artificial neural network with multiple hidden layers between the input and output layers. DNN architectures, e.g., for object detection and parsing, generate compositional models where the object is expressed as a layered composition of image primitives. The extra layers enable composition of features from lower layers, giving the potential of modeling complex data with fewer units than a similarly performing shallow network.

In particular, the kind of neural networks that are used in this project are called feedforward neural networks. This means there are no loops in the network—information is always fed forward, never fed back. DNNs are typically designed as feedforward networks, although research has very successfully applied recurrent neural networks, especially Long Short-Term memory (LSTM), for applications such as language modeling.

A DNN can be discriminatively trained with the standard backpropagation algorithm. The backpropagation algorithm has had quite a story but, long story short, the modern version of the algorithm is based on the general method for automatic differentiation (AD) published by Seppo Linvainmaa in 1970. The weight updates of backpropagation can be done via Stochastic Gradient Descent (SGD).

The algorithm repeats a two phase cycle, propagation and weight update. When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. The output of the network is then compared to the desired output, using a loss function, and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which indicates how to update the weights of the connections in the network.

Backpropagation uses these error values to calculate the gradient of the loss function with respect to the weights in the network. In the second phase, this gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function.

The importance of this process is that, as the network is trained, the neurons in the intermediate layers organize themselves in such a way that the different neurons learn to recognize different characteristics of the total input space. After training, when an arbitrary input pattern is present which contains noise or is incomplete, neurons in the hidden layer of the network will respond with an active output if the new input contains
a pattern that resembles a feature that the individual neurons have learned to recognize during their training.

On a high level, working with DNNs is a two-stage process: First, a neural network is trained, i.e. its parameters are determined using labeled examples of inputs and desired output. Then, the network is deployed to run inference, using its previously trained parameters to classify, recognize, and generally process unknown inputs.

![Figure 4.2: Deep learning training compared to inference. In training, many inputs, often in large batches, are used to train a deep neural network. In inference, the trained network is used to discover information within new inputs that are fed through the network in smaller batches.](image)

In general, we might say that the per-image workload for training is higher than for inference, and while high throughput is the only thing that counts during training, latency becomes important for inference as well [43].

### 4.1.2 Low-power GPUs and Deep Neural Networks

It is widely recognized within academia and industry that GPUs are the state of the art general purpose processor in training deep neural networks, due to both speed and energy efficiency advantages compared to more traditional CPU-based platforms. Neural networks rely heavily on matrix math operations and complex multi-layered networks require tremendous amounts of floating point performance and memory bandwidth for both efficiency and speed. Because neural networks are created from large numbers of identical neurons, they are highly data parallel by nature. This parallelism maps naturally to GPUs, which provide a significant speed-up over CPU-only training.

In this project a low-power, mobile GPU is used to perform such tasks. While mobile GPUs have improved significantly in computation power in recent years and are capable of carrying out billions of arithmetic computations every second, they also have various resource constraints like power, memory, and compute capability. The purpose of this project is to mitigate the impact of these constraints on DNNs inference performance.

In particular, the experimental setup of this project is based on the Nvidia Tegra X1. This mobile GPU has a Nvidia Maxwell architecture with 256 CUDA cores, 512 GFLOPS for FP32 and 1024 GFLOPS for FP16. Despite not having a Pascal architecture, it does support FP16 arithmetic; this means that operations with half-precision
floating point data types can be performed natively on the GPU, without the need of converting back and forth between FP16 and FP32. Nvidia claims that FP16 arithmetic delivers up to 2x the performance of equivalent FP32 arithmetic. The Nvidia Shield tablet, the Nvidia Jetson TX1 development board, the Google Pixel C or the more recent Nintendo Switch are devices powered by this GPU [44].

In fact, this mobile-centric GPU feature has motivated the first optimization that is carried out in this project: adapting the Caffe framework to use FP16 instead of FP32. In addition to benefit from the FP16 arithmetic feature, it also reduces the memory bandwidth required to do training and inference, since the network only takes half the space in memory.

Memory bandwidth has long been a bottleneck at higher performance levels in low-power GPUs, and while it is a solvable problem, the general solution is to build a wider (96-bit or 128-bit) memory bus, which is very effective but also drives up the cost and complexity of the SoC and the supporting hardware. In this case Nvidia is sticking to a 64-bit memory bus, so memory compression is very important. This fact motivates the second optimization of this project: neuron-pruning. By doing neuron-pruning, the dimensionality of the neural network parameters is reduced, thus reducing the memory bandwidth required to do training and inference [45].

Furthermore, these optimizations greatly reduce the memory space taken by the neural network. As a consequence, the power consumption required to do training and inference is also reduced, which is another of the constraints of low-power GPUs.

4.1.3 Caffe, a deep learning framework

Caffe is a deep learning framework developed by Berkeley AI Research (BAIR) and by community contributors. Yangqing Jia created the project during his PhD at UC Berkeley. The philosophy of the framework is based on expression, speed, modularity, openness and community. These principles direct the project [14].

Deep networks are compositional models that are naturally represented as a collection of inter-connected layers that work on chunks of data. Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss.

As data and derivatives flow through the network in the forward and backward passes Caffe stores, communicates, and manipulates the information as blobs: the blob is the standard multidimensional array and unified memory interface for the framework. The layer comes next as the foundation of both model and computation. The net follows as the collection and connection of layers.

Caffe stores and communicates data using blobs. Blobs provide a unified memory interface holding data; e.g., batches of images, model parameters, and derivatives for optimization.

Blobs conceal the computational and mental overhead of mixed CPU/GPU operation by synchronizing from the CPU host to the GPU device as needed. Memory on the host and device is allocated on demand (lazily) for efficient memory usage.

The conventional blob dimensions for batches of image data are number N x channel K x height H x width W. Blob memory is row-major in layout, so the last / rightmost
dimension changes fastest. For example, in a 4D blob, the value at index \((n, k, h, w)\) is physically located at index \(((n \ast K + k) \ast H + h) \ast W + w\).

As we are often interested in the values as well as the gradients of the blob, a Blob stores two chunks of memories, \textit{data} and \textit{diff}. The former is the normal data that is passed along, and the latter is the gradient computed by the network.

Furthermore, as the actual values could be stored either on the CPU and on the GPU, there are two different ways to access them: the \textit{const} way, which does not change the values, and the \textit{mutable} way, which changes the values:

\begin{verbatim}
const Dtype* cpu_data() const;
Dtype* mutable_cpu_data();
\end{verbatim}

(similarly for \textit{gpu} and \textit{diff}).

The reason for such design is that a Blob uses a \textit{SynchronizedMem} class to synchronize values between the CPU and GPU in order to hide the synchronization details and to minimize data transfer, which is one of the most expensive operations when working with GPUs.

The layer is the essence of a model and the fundamental unit of computation. Layers convolve filters, pool, take inner products, apply nonlinearities like rectified-linear and sigmoid and other elementwise transformations, normalize, load data, and compute losses like softmax and hinge.

A layer takes input through \textit{bottom} connections and makes output through \textit{top} connections. Layers have two key responsibilities for the operation of the network as a whole: a \textit{forward pass} that takes the inputs and produces the outputs, and a \textit{backward pass} that takes the gradient with respect to the output, and computes the gradients with respect to the parameters and to the inputs, which are in turn back-propagated to earlier layers.
The net is a set of layers connected in a computation graph – a directed acyclic graph (DAG) to be exact. A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

The net is defined as a set of layers and their connections in a plaintext modeling language. A simple logistic regression classifier is defined by

```plaintext
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
```

CHAPTER 4 DESIGN AND IMPLEMENTATION
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}

The models are defined in plaintext protocol buffer schema (prototxt) while the learned models are serialized as binary protocol buffer (binaryproto) .caffemodel files. The model format is defined by the protobuf schema in caffe.proto.

Caffe speaks Google Protocol Buffers [46], Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data; similar to XML but smaller, faster and simpler. Protocol buffers produce minimal-size binary strings when serialized, they have a human-readable text format compatible with the binary version and efficient interface implementations in multiple languages, most notably C++ and Python.

Further, Caffe uses cuBLAS [47] in the GPU version of the layers, which is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA runtime. Additionally, the GPU implementation of some Caffe layers uses cuDNN [48], a GPU-accelerated library of primitives for deep neural networks which provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

4.1.4 Faster R-CNN, a very deep neural network

Image recognition encompasses many different problems, being classification and detection two of the most important ones. On one hand, classification consists in predicting the presence or absence of a particular object in an image. On the other hand, detection consists in predicting the bounding box and label of each of the objects that appear in the image. In this project, we are trying to optimize an object detection problem.

An object detection system is usually composed by two elements: a region proposal method to hypothesize object locations and a detection network to predict the content of those locations. Recent advances in object detection are driven by the success of region proposal methods [49] and region-based CNNs [32]. Nowadays, advances in region-based CNNs have exposed the computation of region proposals as the main bottleneck when performing inference computations. In the past, region proposal methods such as Selective Search [49] or EdgeBoxes [50] relied on inexpensive features and economical inference schemes. Nevertheless, using these methods the region proposal step still consumes as much running time as the detection network.

The object detection system analyzed in this project, called Faster R-CNN, is composed of two modules. The first module is a deep convolutional network that proposes regions, and the second module is the Fast R-CNN detector [32] that identifies the objects in the proposed regions. The RPN module tells the Fast R-CNN module where to look. The entire system is a single, unified network for object detection (Figure 4.3).
A Region Proposal Network (RPN) takes an image as input and outputs a set of rectangular object proposals, each with an objectness score\footnote{"Objectness" measures membership to a set of object classes vs. background.}. This process is modeled with a convolutional network, which is described in this section. Because the goal of the RPN architecture is to share computation with a Fast-RCNN object detection network, both nets share a common set of convolutional layers. In this project, the experiments are done with the Simonyan and Zisserman model (VGG16)\cite{51}, which has 13 shareable convolutional layers.

To generate region proposals, a small network is slid over the convolutional feature map output by the last shared convolutional layer. This small network takes as input an $n \times n$ spatial window of the input convolutional feature map. At each sliding-window location, multiple region proposals are simultaneously predicted, where the maximum number of possible proposals for each location is denoted as $k$. The $k$ proposals are parametrized relative to $k$ reference boxes called anchors. An anchor is centered at the sliding window in question, and is associated with a scale and aspect ratio. By default, 3 scales and 3 aspect ratios are used, yielding $k = 9$ anchors at each sliding position. For a convolutional map of a size $W \times H$, there are $WHk$ anchors in total.

For the detection network, Fast R-CNN\cite{32} is used. A Fast R-CNN network takes as input an entire image and a set of object proposals (provided by the RPN). The network first processes the whole image with several convolutional and max pooling layers to produce a conv feature map. Then, for each object proposal a region of interest (RoI) pooling layer extracts a fixed-length feature vector from the feature map. Each feature vector is fed into a sequence of fully connected layers that finally branch into two sibling output layers: one that produces softmax probability estimates over $K$ object classes plus a "background" class and another layer that outputs four real-valued numbers for each of the $K$ object classes. Each set of 4 values encodes refined bounding-box positions for one of the $K$ classes (Figure 4.4).
In their paper [13], Shaoqing Ren et al. describe several algorithms that learn a unified network composed of RPN and Fast R-CNN with shared convolutional layers (Figure 4.3). In this project, we only use a method called approximate joint training. In this solution, the RPN and Fast R-CNN networks are merged into one network during training as in figure 4.3. In each Stochastic Gradient Descent iteration, the forward pass generates region proposals which are treated just like fixed, pre-computed proposals when training a Fast R-CNN detector. The backward propagation takes place as usual, where for the shared layers the backward propagated signals from both the RPN loss and the Fast R-CNN loss are combined.

The accuracy of this object detection system is evaluated on the Pascal VOC 2007 detection benchmark [52]. This dataset consists of about 5k trainval images and 5k test images over 20 object categories. The evaluation of the detection results is primarily focused on mean Average Precision (mAP), because this is the actual metric used in the object detection Pascal VOC challenge.

For the VOC2007 challenge, the interpolated average precision [53] was used to evaluate both classification and detection. For a given task and class, the precision/recall curve is computed from a method’s ranked output. Recall is defined as the proportion of all positive examples ranked above a given rank. Precision is the proportion of all examples above that rank which are from the positive class. The AP summarizes the shape of the precision/recall curve, and is defined as the mean precision at a set of eleven equally spaced recall levels [0, 0.1, . . . , 1]:

\[
AP = \frac{1}{11} \sum_{r \in \{0,0.1,...,1\}} p_{\text{interp}}(r)
\]  

(4.1)

The precision at each recall level r is interpolated by taking the maximum precision measured for a method for which the corresponding recall exceeds r:

\[
p_{\text{interp}}(r) = \max_{\tilde{r} : \tilde{r} \geq r} p(\tilde{r})
\]  

(4.2)

where \(p(\tilde{r})\) is the measured precision at recall \(\tilde{r}\).

It should be noted that to obtain a high score, a method must have precision at all levels of recall – this penalizes methods which retrieve only a subset of examples with high precision (e.g. side views of cars).
In the challenge, detections were assigned to ground truth objects and judged to be true/false positives by measuring bounding box overlap. To be considered a correct detection, the overlap ratio \(a\) between the predicted bounding box \(B_p\) and ground truth bounding box \(B_{gt}\) must exceed 50% by the formula:

\[
a = \frac{\text{area}(B_p \cap B_{gt})}{\text{area}(B_p \cup B_{gt})}
\]  

where \(B_p \cap B_{gt}\) denotes the intersection of the predicted and ground truth bounding boxes and \(B_p \cup B_{gt}\) their union.

Although the official Faster-RCNN code is written in Matlab, the Python reimplementation is used in this project. The Faster-RCNN application runs on top of the Caffe deep learning framework through Boost Python, since Caffe is written in C++ but Faster R-CNN is written in Python. The Boost Python Library is a framework for interfacing Python and C++. Basically, it allows to expose C++ classes, functions and objects to Python through a library imported in the Python code. The _caffe.cpp file inside caffe-fast-rcnn/python/caffe contains the wrappers that expose the Caffe C++ classes and methods to Python. Then, this file is compiled into the dynamic library _caffe.so, which can be imported from the Python code.

The released Python code contains pre-computed Faster R-CNN detectors trained on VOC 2007 trainval, pre-trained ImageNet models to initialize the network weights and the implementation of the approximate joint training and alternating optimization training methods. In addition, it includes several scripts to train and test the resulting model following the Pascal VOC 2007 evaluation method.

### 4.2 Initial neural network characterization

#### 4.2.1 Network anatomy

##### 4.2.1.1 Types of layers

**Convolutional layer** The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. Throughout this document, the conv prefix refers to a convolutional layer.

The input of fully-connected layers is effectively a 3D array. The image-to-column (im2col) function will rearrange the 3D array into a 2D matrix (input matrix). Then, the weights matrix\(^2\) will be multiplied with the input matrix to generate an output matrix which can be converted back to a 3D array for the next layer (Figure 4.5).

\(^2\)The weights of each layer can be grouped into a weight matrix.
Figure 4.5: Matrix-matrix multiplication in a convolutional layer. Here $k$ is the number of values in each patch and kernel, so it’s kernel\_width \times kernel\_height \times depth, $m$ is the number of patches, and $n$ is the number of kernels. $A$ is the input matrix, $B$ is the weights matrix and $C$ is the output matrix.

**Pooling layer** It is common to periodically insert a pooling layer in-between successive convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. A pooling layer operates independently on every depth slice of the input and resizes it spatially, using the max operation. Throughout this document, the pool prefix refers to a pooling layer.

**Fully-connected layer** Neurons in a fully-connected layer have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset. Throughout this document, the fc prefix refers to a fully-connected layer.

In fully-connected layers, the input values are stored in a 1D vector (input vector). Then, the weights matrix can be multiplied with the input vector to generate the output which is also a 1D vector (Figure 4.6).

Figure 4.6: Matrix-vector multiplication in a fully-connected layer. There are $k$ input values, and there are $n$ neurons, each one of which has its own set of learned weights for every input value.
4.2.1.2 Network topology

The particular network being analyzed in this project is a combination of a Fast R-CNN network and an RPN. More specifically, the version of the Fast R-CNN network that is being used is called VGG16, and is a modified version of the network developed by the Visual Geometry Group from the University of Oxford [56]. Appendix A shows a summarized diagram of the network topology.

As stated before, both networks share the first 13 convolutional layers (from conv1_1 to conv5_3).

Then, the convolutional feature map output by the last convolutional layer (conv5_-3) is duplicated and fed to the last part of both networks. The RPN takes as input a 3 × 3 spatial window of the input convolutional feature map, and 512 filters encoding lower-dimensional features are applied to each window (rpn_conv/3x3). Then, these features are fed into two sibling fully-connected layers, a box-regression layer (reg) and a box-classification layer (cls). The result are bounding boxes, each one with a score for each of the 20 possible classes.

Finally, the Fast R-CNN network takes as input the convolutional feature map output by the last shared convolutional layer (conv5_3) and also the region proposals output from the RPN. Then, for each object proposal a region of interest (RoI) pooling layer (roi_pool5) extracts feature vectors from the feature map and each vector is fed into a sequence of fully connected layers (fc6 and fc7), which finally branch into two sibling output layers (cls_score and bbox_pred).

Table 4.1 shows the shape of each blob, which can be an input, an output or both. Usually, the blob has the same name as the layer from which it is output. As a remainder, each blob has 4 dimensions (maybe less), which correspond to (number, channel, height, width). Here the prefix rpn means that the blob belongs to the Region Proposal Network.

<table>
<thead>
<tr>
<th>Blob</th>
<th>Shape</th>
<th>Blob</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>(1, 64, 224, 224)</td>
<td>conv5_3</td>
<td>(1, 512, 14, 14)</td>
</tr>
<tr>
<td>conv1_2</td>
<td>(1, 64, 224, 224)</td>
<td>conv5_3_relu5_3</td>
<td>(1, 512, 14, 14)</td>
</tr>
<tr>
<td>pool1</td>
<td>(1, 64, 112, 112)</td>
<td>rpn/output</td>
<td>(1, 512, 14, 14)</td>
</tr>
<tr>
<td>conv2_1</td>
<td>(1, 128, 112, 112)</td>
<td>rpn/output_rpn_relu/3x3</td>
<td>(1, 512, 14, 14)</td>
</tr>
<tr>
<td>conv2_2</td>
<td>(1, 128, 112, 112)</td>
<td>rpn_cls_score</td>
<td>(1, 18, 14, 14)</td>
</tr>
<tr>
<td>pool2</td>
<td>(1, 128, 56, 56)</td>
<td>rpn_bbox_pred</td>
<td>(1, 36, 14, 14)</td>
</tr>
<tr>
<td>conv3_1</td>
<td>(1, 256, 56, 56)</td>
<td>rpn_cls_score_reshape</td>
<td>(1, 2, 126, 14)</td>
</tr>
<tr>
<td>conv3_2</td>
<td>(1, 256, 56, 56)</td>
<td>rpn_cls_prob</td>
<td>(1, 2, 126, 14)</td>
</tr>
<tr>
<td>conv3_3</td>
<td>(1, 256, 56, 56)</td>
<td>rpn_cls_prob_reshape</td>
<td>(1, 18, 14, 14)</td>
</tr>
<tr>
<td>pool3</td>
<td>(1, 256, 28, 28)</td>
<td>rois</td>
<td>(1, 5)</td>
</tr>
<tr>
<td>conv4_1</td>
<td>(1, 512, 28, 28)</td>
<td>pool5</td>
<td>(1, 512, 7, 7)</td>
</tr>
<tr>
<td>conv4_2</td>
<td>(1, 512, 28, 28)</td>
<td>fc6</td>
<td>(1, 4096)</td>
</tr>
<tr>
<td>conv4_3</td>
<td>(1, 512, 28, 28)</td>
<td>fc7</td>
<td>(1, 4096)</td>
</tr>
<tr>
<td>pool4</td>
<td>(1, 512, 14, 14)</td>
<td>cls_score</td>
<td>(1, 21)</td>
</tr>
<tr>
<td>conv5_1</td>
<td>(1, 512, 14, 14)</td>
<td>bbox_pred</td>
<td>(1, 84)</td>
</tr>
<tr>
<td>conv5_2</td>
<td>(1, 512, 14, 14)</td>
<td>cls_prob</td>
<td>(1, 21)</td>
</tr>
</tbody>
</table>

Table 4.1: Shape per blob.
Notice that the first dimension is always 1. This is because here the input was a single image.

### 4.2.2 Network profiling

Since the optimizations are focused on reducing the execution time and the memory footprint of the inference step, we are interested in which layers take longer to execute and which ones require more memory. The following results were taken on a Nvidia Jetson TX1 module.

In figure 4.7 we can observe the average execution time per layer, and in appendix B we can find the detailed numbers. These numbers were obtained by averaging the results of the forward pass on 10 different images.

![Figure 4.7: Average execution time per layer.](image)

As we can see, the fully-connected layer \( fc6 \) alone takes up to 40% of the execution time, while the convolutional and fully-connected layers sum up to 88.16% of the total execution time.

Figure 4.8 shows how much memory take the weights in each layer of the network. The weights of the connections between neurons are the *learnable* parameters of the neural network. Not all layers have weights, so in figure 4.8 only layers whose connections have weights are shown. Appendix C shows the detailed numbers.

In total, the model `VGG16_faster_rcnn_final.caffemodel` takes 548.313 MB of space, which is more than half a GB. We observe that the layer \( fc6 \) alone takes up to
74.97% of the memory, while the two fully-connected layers, fc6 and fc7, sum up to 87.2% of the total memory taken by the learnable parameters of the neural network.

The obtained results are consistent. The majority of the execution time is spent in the convolutional and fully connected layers, whose computation is based on matrix-vector and matrix-matrix products, which are very expensive operations. Furthermore, fully-connected layers are the most expensive in terms of memory footprint since they connect all input neurons to all output neurons, and there is a weight for each of those connections, which is stored with a float (4 bytes).

In conclusion, the main performance and memory bottleneck are convolutional and fully-connected layers. Therefore, these layers will be the focus of this project.

### 4.3 FP16 Optimization

This optimization consists in adapting the Caffe framework to use half-precision floating-point data types instead of single-precision. It is aimed to exploit the FP16 units in the Nvidia Tegra X1, which allow to perform FP16 arithmetic natively on the GPU, delivering up to 2x the throughput of equivalent FP32 arithmetic according to Nvidia. Furthermore, this optimization reduces by half the memory footprint of the weights once they are loaded in memory, since half-precision floats are stored in 16 bits instead of 32 bits. As a consequence, using a half-precision representation saves energy in the memory and communication channels, improves performance in memory bound systems.
through better memory bandwidth utilization and effective cache capacity, and supports larger networks on systems with a fixed memory budget.

The source code of this optimization can be found in: [https://github.com/oscmansan/py-faster-rcnn-fp16](https://github.com/oscmansan/py-faster-rcnn-fp16) and [https://github.com/oscmansan/caffe-fast-rcnn/tree/fp16](https://github.com/oscmansan/caffe-fast-rcnn/tree/fp16).

### 4.3.1 Half-precision floating-point format

In IEEE 754 the half precision format is officially referred to as binary16, since it occupies 16 bits. It is intended for storage of many floating-point values where higher precision is not needed, not for performing arithmetic computations.

The standard specifies a binary16 as having the following format: 1 bit for the sign, 5 bits for the exponent and 11 bits for the mantissa (10 explicitly stored). The format is laid out as follows:

*Figure 4.9: IEEE 754 half-precision binary floating-point format.*

The format is assumed to have an implicit lead bit with value 1 unless the exponent field is stored with all zeros. Thus, only 10 bits of the mantissa appear in the memory format but the total precision is 11 bits. In IEEE 754 parlance, there are 10 bits of mantissa, but there are 11 bits of mantissa precision [57].

Using FP16 has been proven to achieve the same classification accuracy as FP32. In [34], Gupta et al. explore the effect of employing limited precision data representation during the training of deep neural networks. For low-precision computations, they adopt stochastic rounding during deep neural network training, which delivers results nearly identical as FP32 computations. Neural networks are known to be error tolerant, so they do not require 32-bit precision to achieve good prediction accuracy.

### 4.3.2 Implementation

I considered several approaches to adapt the Caffe framework to be able to run with FP16. Nvidia maintains a fork of Caffe tuned for Nvidia GPUs [58], which has support for 16-bit (half) floating point training and inference. But Faster R-CNN uses its own fork of Caffe, which supports Fast R-CNN [59]. In the first moment, I thought that changing the caffe-fast-rcnn submodule of the Faster R-CNN repository for nvcaffe would suffice to run Caffe with half-precision floats. It turns out that this is not enough at all. The application Faster R-CNN needs some layers that do not exist in the original Caffe, so the Nvidia fork of Caffe does not have these layers either. For example, the RoI pooling layer, which uses max pooling to convert the features inside any RoI into a small feature map, was implemented in caffe-fast-rcnn but not in nvcaffe.
There were two possible approaches. On one hand, I could start from the Nvidia Caffe fork and gradually add and adapt all the extra files of `caffe-fast-rcnn` that Faster R-CNN needed to work. On the other hand, I could start from the Fast R-CNN Caffe fork and gradually adapt all the required files to run with FP16, based on the work of Nvidia folks. After thorough consideration I decided to take the second approach, since the code was already working with FP32.

The first step was to copy some files from the Nvidia fork of Caffe which are necessary for the conversion from/to FP16. `get.hpp` provides the method `Get<T>(const Y& y)`, which converts from type `Y` to type `T`. Here is an example of conversion from `float` to `float16` and vice versa:

```cpp
float x = N;
float16 y = Get<float16>(x);
float z = Get<float>(y);
```

`get.hpp` includes `float16.hpp`, which calls `__float2half_rn` if Caffe is compiled with GPU support or `cpu_float2half_rn` otherwise. The first one is included from `cuda_fp16.h`, which is included from `fp16_emu.h`, and the second one is included in `fp16_conversion.hpp` and implemented in `fp16_conversion.cu`. These are all the files needed to use the type `float16` in Caffe.

Next I started modifying, one by one, all the Caffe files required for Faster R-CNN to work. Further, I wrote a new Makefile to build only the files I was modifying (the original Makefile compiled all the classes inside the project). Otherwise, I would have had conflicts during compilation because some modified classes were dependency of other classes that were not necessary for the application to work. In order to avoid useless work, I only modified the necessary classes, and thus the signature of some methods differed across files. If I had compiled all the classes, I would have run into conflicts during compilation.

The Caffe framework uses C++ templates to make the implementation generic and independent from the type of data stored in data structures. For example, the signature of the class `Blob` is as follows:

```cpp
template<typename Dtype>
class Blob {
    ...
}
```

Here, `Dtype` means that the blob contains data of type `Dtype`, which can be `int`, `float`, `double`, etc. Nevertheless, the Nvidia version of Caffe uses C++ templates as follows:

```cpp
template<typename Dtype, typename Mtype>
class Blob {
    ...
}
```

That is because the underlying hardware may have support for FP16 storage but not for FP16 arithmetic. In such cases, the data is stored in 16 bits, but it needs to

---

*CHAPTER 4. DESIGN AND IMPLEMENTATION*
be converted to 32-bit floats before operation. An example of this can be found in the `caffeiscal` function inside `math_functions.cpp`:

```c
void caffeiscal<float16, float>(const int N,
    const float alpha, float16 *X) {
    for (int i = 0; i < N; ++i) {
        X[i] = Get<float16>( alpha * Get<float>(X[i]) );
    }
}
```

Therefore, in order to use `float16` in Caffe, I changed the template signature of all the Caffe files required by Faster R-CNN to accept two parameters, one for `Dtype` and the other for `Mtype`. Furthermore, it is possible to set default values for template parameters. This avoided me having to change instantiations of classes whose signature I had modified and now had an extra template parameter. For example, if I have the following signature for the class `Blob`:

```c
template <typename Dtype, typename Mtype = Dtype>
class Blob {
...
}
```

the following instantiations are equivalent:

```c
Blob* blob = new Blob<float16, float16>();
Blob* blob = new Blob<float16>();
```

On the other hand, the files `math_functions.cpp` and `math_functions.cu` implement functions with explicit types and, therefore, they also had to be adapted. Hence, I also added the `float16` version of `caffe_gpu_gemm`, `caffe_gpu_gemv`, `caffe_gpu_axpy`, and the rest of math-related functions that wrap the linear algebra operations of cuBLAS. Again, in order to avoid useless work, I only implemented the GPU version of the functions, since I knew for sure the experiments were going to be run on a GPU. In order to implement the half-precision version of these functions, I used the corresponding version of cuBLAS calls. For example, for `caffe_gpu_gemm` I had to replace the `cublasSgemm` call of the `float` version

```c
void caffe_gpu_gemm<float, float>(...) {
...
    cublasSgemm(...);
}
```

with the `cublasHgemm` call

```c
void caffe_gpu_gemm<float16, float16>(...) {
...
    cublasHgemm(...);
}
```
Here, $S$ stands for single whereas $H$ stands for half.

These are the highlights of the changes I made in the Caffe code. In total, I made 4873 insertions and 1441 deletions over 85 files. To put those numbers in context, the caffe_fast_rcnn repository has 619 files and almost 176,000 lines of code. This may give an idea of the magnitude of the Caffe codebase, and thus the difficulty of the changes in the code needed to implement this optimization. It was out of the scope of this project to have a deep knowledge of what every line of code in Caffe does. Nevertheless, a general knowledge was required to know which files needed to be modified in order to make Caffe work with float16.

### 4.3.3 Unit tests per layer

As stated before, the layer is the essence of a model and the fundamental unit of computation in Caffe. Thus, if a layer does not work properly, the neural network will not give a correct result. Therefore, it was essential to make sure that every layer in the net gave the same result with FP32 and FP16 (except for minor precision errors introduced by the conversion).

With that purpose in mind, I implemented a series of unit tests, one per layer. In order to ease my debug, I ran each test right after implementing the corresponding layer, so that the possible bugs I may had introduced were easier to locate and solve.

In Caffe, layers are the building blocks of a net. They are created from a Layer-Factory, which allows to register layers. Then, during runtime, registered layers can be called by passing a LayerParameter protobuf to the CreateLayer function:

```cpp
LayerRegistry<Dtype>::CreateLayer(param);
```

This way, a layer can be created from a simple string. First, Caffe parses a .prototxt file using the function Message.ParseFromString from the Protobuf C++ API. This returns a NetParameter object, which contains an array of LayerParameter objects. These LayerParameter objects are then passed, one by one, to the CreateLayer function, which returns Caffe.Layer objects. This methodology allows Caffe to create a net and automatically set up all its layers from a simple model file.

The aim of unit tests, though, is to test a piece of code (a layer in this case) independently from the rest of the program. Thus, the first challenge was to construct a layer in isolation, without using the LayerFactory class. Here is an example with the unit test of the ReLULayer.

The first step is to set up the bottom (input) and top (output) blobs. In fact, a layer works with vectors of blobs; this is because a layer can have multiple inputs from more than one layer.

```cpp
Blob<Dtype,Mtype>* bottom_blob;
Blob<Dtype,Mtype>* top_blob;
vector<Blob<Dtype,Mtype>> bottom;
vector<Blob<Dtype,Mtype>> top;
```
// Create bottom blob
bottom_blob = new Blob<Dtype,Mtype>();
blob.push_back(bottom_blob);

// Reshape bottom blob
vector<int> shape {num, channels, height, width};
blob->Reshape(shape);

// Create top blob
top_blob = new Blob<Dtype,Mtype>();
top.push_back(top_blob);

Also, the input blob needs to be reshaped. When Caffe is building the net, the input of a layer is reshaped with the same shape as the output of the previous layer. Since in the unit test the layer is isolated, the shape of the input blob needs to be specified.

The next step is to fill the input blob with random data. Fortunately, Caffe has a Filler class which does exactly that. Actually, Filler is an abstract class that needs to be extended to implement the method Fill. The possible implementations are ConstantFiller, UniformFiller, GaussianFiller, PositiveUnitballFiller, XavierFiller, MSRAFiller and BilinearFiller. In this test I used the GaussianFiller, which fills a Blob with Gaussian-distributed random values; for the purpose of testing, though, all fillers were equivalent. Internally, fillers are initialized with the same random seed in order to generate comparable results.

// Fill bottom blob
FillerParameter filler_param;
GaussianFiller<Dtype,Mtype> filler(filler_param);
filler.Fill(bottom_blob);

Once the input and output blobs are prepared, the layer can be created. It takes as input a LayerParameter protobuf, which can be empty in this case. Once it is created, the method SetUp needs to be called; this method checks that the number of bottom and top blobs is correct, calls LayerSetUp to do one-time layer specific setup, and finally calls Reshape to set up sizes of top blobs and internal buffers.

// Set up layer parameters
LayerParameter layer_param;

// Create layer
RelULayer<Dtype,Mtype> layer(layer_param);
layer.SetUp(bottom,top);

Now the layer is prepared to run the forward pass, which is the method that we actually want to test.

// Run forward pass
layer.Forward(bottom,top);
After that, we need to check that the output values are correct. In this case, since the layer is of type ReLU, the function applied to every input value is \( f(x) = \max(0, x) \). Therefore, we need to check that if an input value was negative, it has been converted to zero in the output, and if it was greater or equal to zero, it has been left unchanged.

```cpp
// Check values
const Dtype* bottom_data = bottom_blob->cpu_data();
const Dtype* top_data = top_blob->cpu_data();
for (int i = 0; i < bottom_blob->count(); ++i) {
    assert(top_data[i] >= 0.);
    assert(top_data[i] == 0. || top_data[i] == bottom_data[i]);
}
```

I made tests for the following layers: ConvolutionalLayer, InnerProductLayer, PoolingLayer, SplitLayer, ReLULayer, SoftmaxLayer, ReshapeLayer, ROI Pooling Layer and PythonLayer. These are the layers used by the deep neural network being analyzed in this project. Due to the nature of the layer, the PythonLayer test requires further explanation.

In the end, all adapted layers passed their corresponding unit tests, verifying that the implementation was correct. Furthermore, we observed that the difference between FP32 and FP16 values was less than \( 10^{-4} \), which makes sense since a half-precision floating point value has only 10 bits of mantissa. Nevertheless, the nature of neural networks makes them very tolerant to small approximation errors in the internal calculations.

### 4.3.3.1 PythonLayer test

The PythonLayer is different from the rest of layers in Caffe. This is because the methods that every layer needs to implement, LayerSetUp, Forward and Backward, are implemented in Python instead of C++, although they are still declared in a C++ header file. This is possible thanks to the Boost Python library, which allows interoperability between both languages. So, at compilation time, the declaration of the functions is taken from `python_layer.hpp` while the implementation is taken form `proposal_layer.py`. The Python implementation uses NumPy data structures to store the input and output blobs. Fortunately, NumPy has a `float16` type, so Boost Python directly translates `Blob<float16, float16>` to a NumPy array with `dtype=float16`.

The ProposalLayer, which is the PythonLayer used by the Faster R-CNN network, basically outputs object detection proposals by applying predicted bounding-box transformations to a set of regular boxes, called anchors. Therefore, there was no way of checking if the layer was giving a correct result other than running some sort of A/B test.

An A/B test is an experimental setup used to evaluate a factor with two levels, a control (version A) and a treatment (version B). The control is usually the default version and the treatment is the change that is being tested. In our case, the control version was the FP32 version of the layer and the treatment version was the FP16 version. The idea was to input the same data to both versions of the layer and see if the output was the same, element by element (again, except for minor precision errors introduced by the conversion).
Because of the kind of computation done in the layer, the input data could not be random. Otherwise, the layer may have not output any proposal (as it actually happened at first), impeding any comparison between versions. This is because in its computation, the layer discards any proposal with a score below a certain threshold. Depending on the range of the random input data, all the scores might be below that threshold, ending all discarded.

The solution was to capture the input and output data from the ProposalLayer during a forward pass in the original net. This was done in the forward method of the Python implementation. This approach, though, required to store multidimensional arrays from Python and be able to read them later and construct C++ structures containing that same data, since the unit test was written in C++. There were several options to serialize data from Python and deserialize it from C++, like XML, Json or Google Protocol Buffers. In the end, we decided to use Json.

The following Python snippet was used to dump the input data of the layer to a file:

```python
with open('bottom','w') as f:
    scores = bottom[0].data.tolist()
    bbox_deltas = bottom[1].data.tolist()
    im_info = bottom[2].data.tolist()
    json.dump([scores,bbox_deltas,im_info],f)
f.close()
```

Once the data was serialized to a file, we needed to read it from the unit test and convert it to the input blob of the layer. To read the Json data in C++, the Boost library was used. Boost provides the method read_json, which converts an input file stream into a property tree. Then, the property tree is traversed to build the Blob object. The following C++ snippet does just that:

```cpp
// Fill bottom blob
ifstream jsonFile("bottom");
ptree pt;
read_json(jsonFile, pt);
int i = 0;
BOOST_FOR_EACH (ptree::value_type const& item, pt) {
    vector<float> vector_data = parse_blob<float>(item.second);
    Dtype* blob_data = bottom[i]->mutable_cpu_data();
    assert(vector_data.size() == bottom[i]->count());
    for (int j = 0; j < vector_data.size(); ++j) {
        blob_data[j] = Get<Dtype>(vector_data[j]);
    }
    ++i;
}
```

The previous code uses the method parse_blob, which traverses a property tree in pre-order and copies the elements to a vector. Then, that vector is converted to a Blob.
template <typename T>
vector<T> parse_blob(ptree const& pt)
{
    vector<T> data;
    BOOST_FOREACH (ptree::value_type const& item, pt) {
        try {
            data.push_back(item.second.get_value<T>());
        }
        catch(exception const& e) {
            vector<T> data = parse_blob<T>(item.second);
            BOOST_FOREACH (T x, data) {
                data.push_back(x);
            }
        }
    }
    return data;
}

One of the flaws of the Boost property tree is that there is no way of knowing if the current subtree is a leaf node. Hence, a small hack was used. If the subtree is a leaf node, the data can be accessed through the get_value<T> method, and the type of data T must be specified. If the subtree cannot be converted to the specified type (as it happens if the data is not a leaf node), the method raises an exception. So, the hack consists in capturing that exception and doing a recursive call inside the catch, since then we know that the subtree is not a leaf and, therefore, it has child nodes.

Finally, the input blob was used to run the forward pass of the layer with FP32 and FP16. Then, both outputs were compared in order to make sure that the FP16 implementation of the PythonLayer was working properly.

4.3.4 Summary of results

The following results were taken on a Nvidia Jetson TX1 module. On average, this optimization achieved a speedup of 1.29x in the forward pass execution time. Also, it reduced the forward pass energy consumption of the whole SoC by 23.55%. Furthermore, the energy-delay of the FP16 version, a metric relating execution time and energy consumption, presents an improvement of 68.44% on average with respect to the FP32 version.

A more comprehensive analysis of the results of this optimization is carried out in section 4.5.2.

4.4 Neuron-pruning optimization

Recent evidence reveals that the network depth is of crucial importance [51]. Deeper models with more parameters greatly improve the accuracy of DNNs. For example,

---

3Here parameters must be understood as learnable parameters, i.e. the weights of the connections between neurons.
AlexNet and VGG employ 61M and 138M parameters, respectively, to do classification on 1000 image categories of ImageNet. Large DNN models with lots of internal redundancy can achieve high accuracy, but at the cost of immense computation and energy requirements. This optimization consists in removing redundant neurons in the neural net with no loss in the prediction accuracy.

Accesses to the off-chip system memory are very expensive from an energy perspective in mobile SoCs. Pruning reduces the memory footprint of the neural network model, reducing memory accesses and hence reducing the energy consumption. Furthermore, mobile GPUs have a much smaller amount of FP units compared to desktop/high-end GPUs. This optimization also reduces the size of the matrices involved in layer computations (see section 4.2.1.1), reducing the number of FP operations and hence reducing the execution time of the forward pass. Therefore, pruning is specially well suited for our target platform, as it reduces both main memory accesses and FP operations.

The source code of this optimization can be found in: https://github.com/oscmanshan/py-faster-rcnn/tree/master

4.4.1 Limitations of weight-pruning on GPU platforms

There are several ways of reducing the number of parameters of a neural network. In the first moment, our idea was to apply weight-pruning, that is, reducing the number of weights between neurons.

However, recent results [35] show that, for many hardware configurations, the performance of the networks after weight pruning is actually worse than before pruning. Pruning hurts performance despite removing an average of 80% of the weights.

To understand these counter-intuitive results, we need to examine weight pruning in more depth as well as the structure of the networks. DNNs essentially consist of two types of layers: fully-connected layers and convolutional layers. They perform matrix-vector and matrix-matrix multiplication respectively, as the weights of each layer can be grouped into a weight matrix (see section 4.2.1.1).

Weight-pruning techniques [18] measure the importance of each weight and remove those deemed unimportant, resulting in both memory storage and computation reductions. After weight pruning, redundant weights and related Multiply and Accumulate (MAC) operations are removed. The matrix computation in the pruned networks becomes sparse, thus the remaining weights are stored in a sparse matrix format.

This sparsity is what often leads to a performance decrease in DNN computation. Sparse weight matrices lose the regular structure of dense matrices, and sparse matrix multiplication needs extra computation to decode the sparse format (CSR). Furthermore, in GPUs the sparse matrix computation cannot make optimal usage of the supported hardware, e.g. memory coalescing. Also, dense matrix optimizations, like matrix tiling, are less effective.

Weight pruning leads to large increases in L1 D-Cache load and store misses due to the sparse representation. With 60% pruning, these overheads are not counter-balanced by the computation reduction, hence a net performance loss is observed. At 80% pruning, the combination of modestly lower memory access overheads and higher reductions in computations results in break-even performance. Pruning rates of more
than 80% are necessary to overcome the memory access overhead for this layer and achieve a net performance gain. However, more than 80% of pruning may not be acceptable as it results in a significant accuracy loss.

These results lead us to rethink this optimization and change the pruning technique.

### 4.4.2 Neuron-pruning

Traditional weight pruning techniques will decrease the performance of all DNN layers on highly-parallel hardware. To avoid this performance decrease, neuron-pruning avoids DNN redundancy by removing entire neurons instead of weights. This method effectively removes entire rows or columns from the matrices, resulting in smaller, dense matrices. Furthermore, the removal of a single neuron implies the removal of the weights of all connections that had an end in that neuron, potentially resulting in a higher pruning rate.

Since there is no sound theoretical guidance of choosing the shape (number of hidden neurones of each layer) or scale to achieve optimal DNN structure, developers normally employ certain level of redundancy to guarantee good prediction performance. The basic assumption of neuron-pruning is that the DNN prior to pruning has redundant neurones and, hence, the same functionality can be achieved using a smaller number of neurons.

There exist in the literature different techniques to decide which neurons should to be removed.

#### 4.4.2.1 Scalpel

The first approach to neuron-pruning consisted in trying to apply a recently published technique called *Scalpel* [35]. It uses mask layers to dynamically find out unimportant neurones and block their outputs. The blocked neurones are removed after the training of mask layers. After removing all redundant neurones, mask layers are removed, and the network is retrained to get the pruned DNN model.

Figure 4.10 gives an example of a mask layer for fully-connected layers.

![Figure 4.10: Mask layer. Neuron A-3 with $\alpha_3 = 0$ can be removed.](image)

The output values of layer $A$ need to go through the mask layer $A'$ before propagated to the next layer. Each neuron in the mask layer holds two parameters $\alpha$ and $\beta$. $\alpha$ is a boolean variable ($\alpha \in \{0, 1\}$) and $\beta$ is a floating number between 0 and 1 ($0 \leq \beta \leq 1$).
Let array $Y$ and $Y'$ to be the output activation array of the original layer $A$ and the mask layer $A'$. For $y_i \in Y$ and $y'_i \in Y'$, we have

$$y'_i = \alpha_i \cdot y_i \quad (4.4)$$

With $\alpha_i$ set to 0, the corresponding neuron can be considered as removed because the output $y'_i$ is fixed to 0. The parameter $\beta_i$ is updated during the training of the mask layer, and the value of $\alpha_i$ is set in function of $\beta_i$.

Nevertheless, this technique was finally discarded due to a lack of details in the article, which forced us to deduce some of these details and thus made a correct implementation very difficult.

### 4.4.2.2 Input-weights norm neuron-pruning

Finally, we tried a technique that focuses on neuron level restructuring of DNN [60]. In this approach, hidden neurons of a fully trained DNN are pruned with certain importance function and the reshaped DNN is retrained using back-propagation.

Neuron-pruning is only applied to hidden layers (all layers except input and output layers). Prior to pruning, it is necessary to evaluate the importance of each hidden neuron. In the paper, several neuron importance functions are proposed. In this project we employ an importance function based on the input-weights norm (inorm), which is simpler to implement and achieves results comparable to other heuristics. inorm measures the importance of a neuron $(l, i)$ by the average L1-norm of the weights of its incoming links, which is formulated as

$$\mathcal{J}_i(l, i) = \frac{1}{N_{l-1}} \sum_{j=1}^{N_{l-1}} |w_{ji}| \quad (4.5)$$

where $(l, i)$ represents the output neuron $i$ of the hidden layer $l$, layer $l$ has $N_{l}$ input neurons and $w_{ji}$ is the transition weight on the arc between neuron $j$ of layer $l-1$ and neuron $i$ of layer $l$. Simply put, the importance of a neuron is the average of its incoming weights in absolute value.

After training, a score is calculated for each hidden neuron using the previous importance function. Then all the neurons are sorted by their scores and neurons with less importance values are removed. Along with the removal of a neuron, all relevant incoming and outgoing links are also removed. It’s worth noting that although every neuron removal will affect the scores of some other neurons, in this approach, all neurons are only examined once with the scores calculated using the original DNN.

Directly pruning hidden neurons results in significant accuracy loss. Hence, the pruned DNN needs to be retrained.

We finally decided to go with this pruning technique due to its simplicity and efficiency in reducing the amount of computations and DNN parameters. Furthermore, all the implementation details are available in [60].
4.4.3 Implementation

As explained before, in Caffe a network model is defined by 2 files: a plaintext protocol buffer schema (prototxt), which defines the net topology, and a binary protocol buffer (caffemodel), which contains the serialized weights of a trained net. In order to implement this optimization, we needed to change the network topology, and thus we needed to change these 2 files. On one hand, we had to modify the prototxt file to reflect the new dimensions of the pruned layers. On the other hand, we had to modify the caffemodel file to remove the weights associated to connections that had an end in one of the pruned neurons.

We started by modifying the caffemodel file. Since we had to work with protocol buffers and we had to manipulate data in matrix form, we decided to write a Python script to that end, because Google Protocol Buffers have Python bindings and we could also take advantage of the NumPy package to manipulate the multi-dimensional data.

In the following lines the script used to perform neuron pruning is explained. The first step is reading the caffemodel file, deserialize it, and construct a \texttt{NetParameter} object with the data contained in the file. The deserialization is done by opening the file in \texttt{rb} mode, which means that the file is read in binary mode. To convert the file string to a \texttt{NetParameter} object, the \texttt{ParseFromString} method from the Protobuf Python API is used.

```python
f = open('VGG16_faster_rcnn_final.caffemodel', 'rb')
net_param = caffe.NetParameter()
net_param.ParseFromString(f.read())
f.close()
```

Next, we convert the data (weights) in each layer of the \texttt{NetParameter} object into a NumPy array in order to manipulate it more easily.

```python
layers = {}
for layer in net_param.layer:
    if layer.name in ['fc6','fc7','cls_score','bbox_pred']:
        blobs = layer.blobs
        data = np.array(blobs[0].data).reshape(blobs[0].shape.dim)
        if len(blobs) == 1:
            layers[layer.name] = data
        elif len(blobs) == 2:
            bias = np.array(blobs[1].data).reshape(blobs[1].shape.dim)
            layers[layer.name] = data, bias
```

As explained before, the weights of a fully-connected layer are organized in a $M \times N$ matrix, where $M$ is the input size and $N$ is the output size. However, in Caffe the weights matrix is transposed. This is done because matrices are stored row-wise, and in a matrix-matrix product, the first matrix is accessed by row and the second matrix is accessed by column. Therefore, the second matrix (the weights matrix) is stored column-wise in order to exploit spatial locality.
The next step is to compute the \( \text{inorm} \) score for each (output) neuron in the layer. Since the weight matrix is of the form \( \text{outputs} \times \text{inputs} \), this is done by computing the average of the absolute value of each row of the weights matrix.

\[
inorm = \text{np.abs}(\text{layers['fc6'][0]}).\text{mean}(\text{axis}=1)
\]

Once we have the \( \text{inorm} \) score of each neuron, we need to sort the neurons by score and remove the ones with smaller importance values. In the code, first an indirect sort is done to compute the score threshold \( \text{th} \) that leaves a \( p \% \) of the values below, where \( p \) is the pruning percentage (e.g. in order to prune 80\% of the weights, \( p = 0.8 \)). Then, we obtain a boolean array \text{keep} of the same size as the number of neurons, with \text{true} in the positions where the neuron score is greater than the threshold and \text{false} otherwise. This way, the NumPy array can be indexed with the \text{keep} array to remove neurons with a score below the threshold.

\[
\text{th} = \text{inorm[np.argsort(inorm)[int(p*len(inorm))]]}
\]

\[
\text{keep1} = \text{inorm}>\text{th}
\]

\[
\text{data} = \text{layers['fc6'][0]}
\]

\[
\text{bias} = \text{layers['fc6'][1]}
\]

\[
\text{layers['fc6']} = \text{data[keep1,:]}, \text{bias[keep1]}
\]

This is done for all the layers that need to be pruned. Next, the NumPy arrays that represent the layer weights are converted back to a \text{NetParameter} object, which is written again to a binary protocol buffer file with the help of the \text{SerializeToString} method from the Protobuf Python API.

\[
\text{for layer in net_param.layer:}
\]

\[
\quad \text{if layer.name in ['fc6', 'fc7', 'cls_score', 'bbox_pred']:
\quad \quad \text{blobs} = layer.blobs
\quad \quad \text{data} = \text{layers[layer.name][0]}
\quad \quad \text{bias} = \text{layers[layer.name][1]}
\quad \quad \text{blobs[0].data[:]} = \text{data.flatten()}
\quad \quad \text{blobs[0].shape.dim[:]} = \text{data.shape}
\quad \quad \text{blobs[1].data[:]} = \text{bias.flatten()}
\quad \quad \text{blobs[1].shape.dim[:]} = \text{bias.shape}
\]

\[
\quad \text{f = open('VGG16_faster_rcnn_final.caffemodel.pruned', 'wb')}
\]

\[
\quad \text{f.write(net_param.SerializeToString())}
\]

\[
\quad \text{f.close()}
\]

According to the analysis of the neural network described in section 4.2, the neuron-pruning was focused on the \text{fc6} and \text{fc7} fully-connected layers, since the initial characterization of the neural network showed that those two layers added up to a 87.2\% of the total memory taken by the weights. As a consequence of the restructuring of the network, the \text{cls_score} and \text{bbox_pred} layers, which take as input the output of the \text{fc7} layer, also needed to be modified. Appendix D shows the complete code for the neuron-pruning script.

The next step was modifying the prototxt file to reflect the dimensions of the pruned layers. In this case, we only had to modify the output dimensions of the fully-connected
layers fc6 and fc7. The original output dimension of the layers was 4096, and the new output dimension was 2048 with 50% of pruning and 819 with 80% of pruning. The code below shows the example of the fc6 layer, with the original output dimension on the left and the new output dimension with 80% of pruning on the right.

```
layer {
  name: "fc6"
  type: "InnerProduct"
  bottom: "pool5"
  top: "fc6"
  ... 
  inner_product_param {
    num_output: 4096
  }
}

layer {
  name: "fc6"
  type: "InnerProduct"
  bottom: "pool5"
  top: "fc6"
  ... 
  inner_product_param {
    num_output: 819
  }
}
```

As stated before, directly pruning hidden neurons results in significant accuracy loss. So we needed to retrain the neural network. First, we tested the network before the retraining to see how much accuracy we lost. We used an script provided with the released Python code, test.py, which tests a Fast R-CNN network on an image database. In this case, we used the VOC2007 dataset and the mean Average Precision as accuracy measure (see section 4.1.4). After evaluating detections for 4952 test images, we got a mAP of 49.08% with 80% of pruning, pretty low compared with the 68.99% obtained with the original model.

To retrain the neural network, we also used a script provided with the released Python code. The script, train.py, acts as a Python wrapper of the Caffe SGDSolver, which optimizes the parameters of a net using stochastic gradient descent (SGD). The script trains the network during 70,000 iterations, and takes a snapshot of the net (i.e. saves the network weights in a caffemodel file) every 10,000 iterations. We leave the rest of the parameters, defined in solver.prototxt, with the initial configuration.

The training lasted for 16-17 hours in a machine with an Intel Core i7-6700K CPU, 64 GB of RAM and an Nvidia GTX 980 Ti GPU, with 2816 CUDA cores and 5632 GFLOPS with single-precision arithmetic.

During the training, it is important that the loss, the error between the prediction and the ground truth, keeps getting smaller and more stable. To see how the loss was evolving in real time, I wrote a small Python script which parsed the output file written by the training script and plotted the loss as a function of the number of iterations. Appendix E shows the full code of the script, and figure 4.11 shows an example of the plot output by the script.

After retraining, we ran again the test script and we obtained a mAP of 68.93% with 50% of pruning and 67.92% with 80% of pruning. Therefore, we obtained almost the same accuracy as the original neural network despite having pruned an 80% of the weights in the fc6 and fc7 layers, which represents almost a 70% of the total number of parameters of the net.
4.4.4 Summary of results

The following results were taken on a Nvidia Jetson TX1 module. On average, this optimization achieved a speedup of 1.22x with 50% of pruning and 1.51x with 80% of pruning in the forward pass execution time. Also, it reduced the forward pass energy consumption of the whole SoC by 16.99% with 50% of pruning and 26.2% with 80% of pruning. Furthermore, the energy-delay presents an improvement of 51.38% with 50% of pruning and 103.37% with 80% of pruning.

In addition, the size of the weights (caffemodel) file is 53.16% the original size with 50% of pruning and 28.02% the original size with 80% of pruning.

A more comprehensive analysis of the results of this optimization is carried out in section 4.5.3.

4.5 Results

4.5.1 Evaluation methodology

In this project, we are interested in reducing execution time and energy consumption. Therefore, those will be the metrics used to evaluate our optimizations.

In order to obtain the execution time of a code region, we use the standard C/C++ timers included in the <sys/time.h> header. The header file <Regions.h> (see appendix F)
allows us to instrument the code by placing two directives at the beginning and at the end of each region of code we want to measure.

```
START_REGION(region_name)
    // Code to evaluate
END_REGION(region_name)
```

Then, the output of the execution is redirected to a file, which is then parsed to obtain the execution time of each region.

Additionally, we measure the energy consumption from hardware counters using profiling tools which are used for research in the ARCO group. To measure the energy consumption of a code region, we use a Python script that launches the program that contains the regions of code we want to measure in a separate thread. During the execution of the program, the script reads from I2C/SMBus chip registers using the `i2cget` Bash command. The registers are polled every 0.05 seconds, returning the CPU power and the GPU power at a given instant of time. This gives us a list of samples, where each sample is composed by a timestamp, the CPU power and the GPU power in that instant of time. Combining this information with the timestamps of the beginning and the end of the regions, we obtain an estimation of the CPU and GPU energy consumption during the execution of a region of code.

For each version of the code, we run the forward pass in 10 different images, repeating the execution 5 times per image. In each execution we obtain elapsed time, CPU energy and GPU energy. Then, for each metric and image, we select the minimum value in those 5 executions. To report an aggregate value for each configuration, we take the arithmetic mean of the 10 per-image minimum values for that configuration.

In terms of hardware, the following results were taken on a Nvidia Jetson TX1 module, which packs a Nvidia Tegra X1 GPU, an ARM Cortex-A57 CPU and 4 GB of RAM LPDDR4 with 25.6 GB/s of bandwidth.

### 4.5.2 FP16 optimization results

#### 4.5.2.1 Execution time per layer

Figure 4.12 shows the execution time (in seconds) of the forward pass for a single image, broken down in the time spent in each layer. Blue bars show times for the original version and red bars show times with the FP16 optimization described in section 4.3.

As observed in the initial characterization of the neural network (see section 4.2), the majority of the time is spent in convolutional and fully-connected layers, especially in the fully-connected layer \( fc6 \). In the original version, this layer takes 0.73 seconds – almost a 40% of the total time, followed by the \( conv2_2 \) layer, which takes 0.09 seconds – a 6% of the total execution time. The other fully-connected layer, the \( fc7 \) takes 0.063 seconds – a 3.44% of the total execution time. On average, the execution time per layer is of 37.25 milliseconds.

After the optimization, the \( fc6 \) layer only takes 0.38 seconds – a 26.16% of the total execution time, experimenting a reduction of almost 50%. The \( fc6 \) layer remains the
most expensive, followed again by the \( \text{conv2}_2 \) layer, which still takes 0.09 seconds – a 10.12% of the total execution time, experimenting almost no reduction. The \( \text{fc7} \) layer only takes 0.025 seconds – a 1.7% of the total execution time, experimenting a reduction of 60%. With the optimization, the average execution time per layer decreases to 29.66 milliseconds.

We observe that the fully-connected layers, \( \text{fc6} \) and \( \text{fc7} \), are the largest beneficiaries of this optimization, while convolutional layers show no improvement. This is because fully-connected layers are bandwidth limited and convolutional layers are computationally limited. This means that the use of FP16 storage, which gives a 2x reduction in memory bandwidth, is sufficient to obtain gains in the fully-connected layers, since in this case the bottleneck is the latency of the accesses to main memory to fetch weights.

Nevertheless, in the convolutional layers the same weights are reused by multiple neurons, so in this case the bottleneck is the compute capability of the FP16 arithmetic units. Nvidia claims that FP16 arithmetic can deliver up to 2x the throughput of equivalent FP32 arithmetic. Nevertheless, empirically we do not see an increase in the throughput, which would entail a decrease in the execution time of the convolutional layers. These results led us to investigate the sources of the low FP16 peak performance.

CUDA offers two types of half-precision floating point data types: \texttt{half} and \texttt{half2} \cite{61}. \texttt{half} is a scalar stored in the low half of a 32-bit register, while \texttt{half2} uses the 32 bits of a register to store two halves, i.e. \texttt{half2} is a vector of two elements of 16 bits (Figure 4.13).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.12.png}
\caption{Execution time per layer applying the FP16 optimization.}
\end{figure}
To achieve 2x performance with FP16 arithmetic, vector instructions must be used. Otherwise, the peak performance with scalar FP16 is limited to the same FP32 performance. Note that with scalar instructions, we still get energy savings, as FP16 arithmetic is cheaper than FP32, and the benefits in memory bandwidth usage.

In the GPU implementation of the forward pass of the convolutional layers, there is a call to the function `cudnnConvolutionForward` of the cuDNN library. It turns out that our version of cuDNN (4.0.7) does not support `half2`, and all the internal computation is done with `half`. Therefore, we do not observe an increase in the performance of the convolutional layers because, although cuDNN uses native half-precision floats, it uses the scalar instructions, which leads to a benefit in energy but not in performance. The latest versions of cuDNN already implement the vectorial instructions (with `half2`). Since further changes to the Caffe code are required to use the new version of cuDNN, we leave this as future work.

### 4.5.2.2 Speedup

Figure 4.14 shows the speedup of the forward pass for 10 different images when applying the FP16 optimization.

We observe that average speedup obtained with this optimization is of about 1.29x. The maximum speedup is of about 1.52x, achieved with the `003317.jpg` input, and the minimum speedup is of about 1.13x, achieved with the `002765.jpg` input. Therefore, we observe a consistent reduction of the execution time across different inputs.

The use of half-precision floating point data types leads to a reduction of the execution time for two reasons. On one hand, since each float is stored with only half of the bits, each access to main memory allows to fetch two times more elements; this reduces the number of accesses to main memory, effectively reducing the time needed to fetch the weights.

### 4.5.2.3 SoC energy

Figure 4.15 shows the normalized System on Chip (SoC) energy (CPU+GPU) of the forward pass for 10 different images when applying the FP16 optimization. Here, we consider both the static and dynamic energy consumption.

---

4By normalized energy we mean the energy consumed after the optimization divided by the energy consumed in the original version.
Figure 4.14: Speedup on different inputs applying the FP16 optimization.

Figure 4.15: Normalized SoC energy on different inputs applying the FP16 optimization.
We observe that the average energy reduction obtained with this optimization is of about 23.55%. The maximum energy reduction is of about 35.2%, achieved with the $000456.jpg$ input, and the minimum energy reduction is of about 13.2%, achieved with the $000446.jpg$ input. Therefore, we observe a consistent reduction of the SoC energy consumption across different inputs.

The use of half-precision floating point data types also leads to a reduction of the energy consumption. Each access to main memory has a cost in terms of energy (dynamic power). Since this optimization reduces the number of accesses to main memory, it also reduces the dynamic energy consumption. In addition, the main memory consumes static power just for storing data during the execution of the forward pass. Since this optimization reduces the execution time of the forward pass, it also reduces the static energy consumption. Further, FP16 arithmetic is cheaper than FP32 arithmetic from an energy point of view.

4.5.2.4 Energy-delay

Figure 4.16 shows the improvement in energy-delay of the forward pass for 10 different images when applying the FP16 optimization. Recall that the energy-delay is a metric that relates the execution time with the energy consumption, and consists in the product of both metrics. The improvement in energy-delay is computed by dividing the energy-delay in the baseline (FP32) by the energy-delay in the FP16 version.
We observe that the average improvement in energy-delay obtained with this optimization is of about 1.68x. The maximum improvement energy-delay is of about 2.25x, achieved with the 003317.jpg input, and the minimum improvement in energy-delay is of about 1.33x, achieved with the 000446.jpg input. Therefore, we observe a reduction of the energy-delay across different inputs, meaning that both the execution time and the energy consumption decrease.

4.5.2.5 Memory footprint

Until now we have been worrying about execution time and energy consumption, which are two of the restrictions that impose mobile platforms. Nevertheless, mobile platforms are also restricted by the amount of memory available. This optimization also reduces the memory footprint of the forward pass of a deep neural network. Since half-precision floating point data types are represented by half of the bits than single-precision, the forward pass of the optimized version only needs half of the memory than the forward pass of the original version.

4.5.3 Neuron-pruning optimization results

In this section we report the results of the neuron-pruning technique described in section 4.4.2.2. Recall that the neuron-pruning has only been applied to the fully-connected layers fc6 and fc7. So in this section, when we talk about a certain percentage of pruning, we mean that percentage of pruning only in the fully-connected layers, which represent a 87.2% of the total weights in the neural network.

4.5.3.1 Execution time per layer

Figure 4.17 shows the execution time (in seconds) of the forward pass for a single image, broken down in the time spent in each layer. Blue bars show times for the original version (with 0% of pruning), red bars show times with 50% of pruning, and yellow bars show times with 80% of pruning in the fully-connected layers.

This optimization is focused on the fc6 and fc7 layers, so the execution time of the forward pass of the rest of the layers remains similar, except for small variations because of measuring on a real system. We observe that in the original version, the fc6 layer takes about 0.75 seconds – a 40% of the total execution time, while the fc7 layer takes about 0.07 seconds – a 3.7% of the total execution time. On average, the execution time per layer is of 36.54 milliseconds.

With 50% of pruning, the fc6 layer only takes about 0.44 seconds, – a 28.6% of the total execution time, experimenting a reduction of about 41%. The fc7 layer takes about 0.024 seconds – about 1.6% of the total execution time, experimenting a reduction of 64.76%. With 50% of pruning, the average execution time per layer decreases to 30.13 milliseconds.

With 80% of pruning, the fc6 layer only takes about 0.16 seconds – a 13.21% of the total execution time, experimenting a reduction of about 78%. The fc7 layer takes about 0.0062 seconds – about 0.5% of the total execution time, experimenting a reduction of
90.81%. With 80% of pruning, the average execution time per layer decreases to 24.7 milliseconds.

This performance improvement is due to the large reduction in memory bandwidth usage achieved with the pruning, as fully-connected layers are known to be memory bandwidth bound.

### 4.5.3.2 Speedup

Figure 4.18 shows the speedup of the forward pass for 10 different images when applying 50% and 80% of neuron-pruning. Blue bars show speedups for a 50% of pruning and yellow bars show speedups for an 80% of pruning. As a consequence of Amdahl’s law, since the fc6 and fc7 layers take about 44% of the execution time in the original version, on average the maximum speedup that can be achieved with 50% of pruning is of 1.28x, and of 1.54x with 80% of pruning. Nevertheless, we can observe inputs achieving more speedup than that because for those inputs the fc6 and fc7 layers may take more than 44% of the total execution time.

We observe that the average speedup obtained with 50% of pruning is of about 1.22x. The maximum speedup is of about 1.39x, achieved with the 003317.jpg input. The minimum speedup, though, is not a speedup but a very slight slowdown, of 0.98x with the 000542.jpg input. Excluding the case of slowdown, we observe a consistent reduction of the execution time across different inputs for the rest of the experiments.
Figure 4.18: Speedup on different inputs applying the neuron-pruning optimization.

With 80% of pruning, we observe that the average speedup is of about 1.51x. The maximum speedup is of about 1.67x, achieved with the 006363.jpg input, and the minimum speedup is of about 1.28x, achieved with the 000542.jpg input. Therefore, with 80% of pruning we also observe a consistent reduction of the execution time across different inputs.

Neuron-pruning leads to a reduction of the execution time. Having less neurons implies having less weights of connections between neurons. This reduces the number of accesses to main memory, effectively reducing the time needed to fetch the weights. In addition, neuron-pruning also reduces the amount of floating point operations.

4.5.3.3 SoC energy

Figure [4.19] shows the normalized SoC energy of the forward pass for 10 different images when applying 50% and 80% of neuron-pruning. Blue bars show normalized energy for a 50% of pruning and yellow bars show normalized energy for an 80% of pruning. Again, we consider both the static and dynamic energy consumption.

We observe that the average energy reduction obtained with 50% of pruning is of about 17%. The maximum energy reduction is of about 38%, achieved with the 000456.jpg input. The minimum energy reduction, though, is not a reduction but a very slight increase, of 2.47% with the 000542.jpg input. These energy results are coherent with the execution time results: the input which takes more time to execute
Figure 4.19: Normalized SoC energy on different inputs applying the neuron-pruning optimization.

is also the one that consumes more energy. Excluding the case of energy increase, we observe a consistent reduction of the SoC energy across different inputs.

With 80% of pruning, we observe that the average energy reduction is of about 26.2%. The maximum energy reduction is of about 37.47%, achieved with the 000456.jpg input, and the minimum energy reduction is of about 17.14%, achieved with the 000542.jpg input. Therefore, with 80% of pruning we also observe a consistent reduction of the execution time across different inputs.

Neuron-pruning also leads to a reduction of the energy consumption for the same reasons that using FP16 does. Each access to main memory has a cost in terms of energy (dynamic power). Since pruning a neuron removes all the memory accesses to fetch its weights from main memory, the dynamic energy consumption is reduced. In addition, the main memory consumes static power just for storing data during the execution of the forward pass. Since this optimization reduces the execution time of the forward pass, it also reduces the static energy consumption. Moreover, pruning a neuron avoids all the corresponding FP operations required to compute the output of the neuron, further reducing the energy consumption.

4.5.3.4 Energy-delay

Figure 4.20 shows the relative energy-delay of the forward pass for 10 different images when applying 50% and 80% of neuron-pruning. Blue bars show speedups for a
We observe that the average improvement in energy-delay obtained with 50% of pruning is of about 1.51x. The maximum improvement in energy-delay is of about 2.01x, achieved with the 006363.jpg input, and the minimum improvement in energy-delay is of about 0.95x, achieved with the 000542.jpg input. Notice that the minimum improvement in energy-delay is below one, meaning that the the energy-delay of the original version is smaller than the energy-delay with 50% of pruning. This is consistent with the results in previous sections, since less execution time and less energy consumption give as a result a smaller energy-delay. Excluding that case, we observe a consistent reduction of the energy-delay across different inputs, meaning that both the execution time and the energy consumption decrease.

With 80% of pruning, we observe that the average improvement energy-delay is of about 2.63x. The maximum improvement in energy-delay is of about 2.55x, achieved with the 006363.jpg input, and the minimum improvement in energy-delay is of about 1.55x, achieved with the 000542.jpg input. Therefore, we observe again a consistent reduction of the energy-delay across different inputs, meaning that both the execution time and the energy consumption decrease. We also see that the reduction is significantly greater when pruning 80% of the weights than when pruning only 50%.

Figure 4.20: Improvement in energy-delay on different inputs applying the neuron-pruning optimization. Here bigger means better.
4.5.3.5 Memory footprint

This optimization also reduces the memory footprint of the forward pass of a deep neural network. Depending on the amount of pruning, the reduction will be greater or smaller. In section 4.2 we saw that the \textit{fc6} and \textit{fc7} layers sum up to 87.2\% of the total memory taken by the learnable parameters of the neural network.

The initial size of the parameters file is 548.3 MBytes. Pruning 50\% of the neurons in the fully-connected layers, the size of the parameters file is 291.4 MBytes, achieving a reduction of 46.84\%. With 80\% of pruning, the size of the parameters file is 153.6 MBytes, achieving a reduction of 71.98\%.

4.5.3.6 Accuracy

Previous subsections show that the greater the amount of neuron-pruning, the greater the reduction in execution time and energy consumption. However, too much pruning can damage the prediction accuracy of the neural network. Figure 4.21 shows the average precision per class when applying 0\%, 50\% and 80\% of neuron-pruning.

Recall that the accuracy is evaluated on the VOC2007 dataset (see section 4.1.4), which consists in 5k test images distributed over 20 object categories or classes. For each class, an average precision is computed. The original version, without pruning, has
a mean average precision of 68.99%. With 50% of pruning, we observe that the neural network has a mean average precision of 68.93%, almost identical to the original version. With 80% of pruning, though, we observe that the mean average precision decreases to 67.92%. This is because we have already removed all the redundant neurons, and we are starting to remove important neurons. If the amount of pruning increases past 80%, the accuracy will keep decreasing. Therefore, we stop the pruning at a point where the accuracy loss is still small.

4.5.4 Combined optimizations results

In sections 4.5.2 and 4.5.3 we have shown individual results for each of the optimizations that have been carried out in the project. In this section, we combine all the optimizations we have implemented to see how much gain we can obtain in total.

4.5.4.1 Execution time per layer

Figure 4.22 shows the execution time (in seconds) of the forward pass for a single image, broken down in the time spent in each layer. Blue bars show times for the original version and red bars show times for the optimized version, using FP16 and 80% of pruning in the fc6 and fc7 layers.

We can observe a remarkable speedup in the fc6 layer, which passes from taking 0.746 seconds to taking only 0.086 seconds. This implies a speedup of 8.68x for this
Figure 4.23: Speedup on different inputs applying FP16 and 80% of neuron-pruning.

layer alone. Even more impressive is the speedup achieved in the $fc_7$ layer, which passes from taking 0.068 seconds to taking only 0.0026 seconds, achieving a speedup of 26x.

We see that the remaining layers experience almost no speedup. This is because, as explained in section 4.5.2.1 with the FP16 optimization we obtain a gain in memory bandwidth but not in throughput, which benefits fully-connected layers but not convolutional layers. In addition, the neuron-pruning optimization targets only fully-connected layers which, as seen in section 4.2, take the majority of the execution time.

Furthermore, we observe that the main bottleneck has changed from the fully-connected layers to the convolutional layers. Now, the convolutional layers take up to a 72.4% of the total execution time, while fully-connected layers only represent an 8% of the total execution time.

4.5.4.2 Speedup

Figure 4.23 shows the speedup of the forward pass for 10 different images when applying FP16 and 80% of pruning in the $fc_6$ and $fc_7$ layers.

We observe that the average speedup obtained when applying all optimizations is of about 1.55x. Notice that the speedup achieved when combining all the optimizations is not the sum of the individual speedups achieved with each optimization. This is because the optimizations are not independent from one another, so there is some overlap. For example, using FP16 we reduce the size of all the weights of the neural net; but then,
when pruning neurons, we remove weights that were already affected by the FP16 optimization.

The maximum speedup is of about 1.73x, achieved with the 003317.jpg input, and the minimum speedup is of about 1.34x, achieved with the 002765.jpg input. Therefore, we observe a consistent reduction of the execution time across different inputs when combining all the optimizations, which is greater than the reduction obtained when applying any optimization individually.

4.5.4.3 SoC energy

Figure 4.24 shows the normalized SoC energy of the forward pass for 10 different images when applying FP16 and 80% of pruning in the fc6 and fc7 layers.

We observe that the average energy reduction obtained when applying all optimizations is of about 31.3%. Again, the energy reduction achieved when combining all optimizations is not the sum of the individual energy reductions achieved with each optimization because there is some overlap between optimizations.

The maximum energy reduction is of about 39.7%, achieved with the 006363.jpg input, while the minimum speedup is of about 21.5%, achieved with the 002765.jpg input. Therefore, we observe a consistent reduction of the SoC energy consumption across different inputs when combining all the optimizations, which is greater than the reduction obtained when applying any of the optimizations alone.
4.5.4.4 Energy-delay

Figure 4.25 shows the improvement in energy-delay of the forward pass for 10 different images when applying FP16 and 80% of pruning in the fc6 and fc7 layers.

![Image of Figure 4.25: Improvement in energy-delay on different inputs applying FP16 and 80% of pruning.]

We observe that the average improvement in energy-delay obtained when applying all optimizations is of about 2.26x. The maximum improvement in energy-delay is of about 2.82x, achieved with the 0006363.jpg input, and the minimum improvement in energy-delay is of about 1.71x with the 002765.jpg input. Therefore, we observe a consistent reduction of the energy-delay across different inputs, meaning that both the execution time and the energy consumption decrease when combining all the optimizations.

4.5.4.5 Memory footprint

Figure 4.26 shows a comparison of the amount of memory (in MB) required by the different versions of the neural network.

The leftmost bar shows the original size of the parameters file, 548.3 MBytes. The next bar shows the size of the parameters file after applying the FP16 optimization, which is exactly half of the size of the original version, 274.1 MBytes. The next bar represents the size of the parameters file after applying a 50% of neuron-pruning, 291.4 MBytes; notice that it is a bit higher than 50% of the original size. This is because we
are pruning a 50% of neurons only in the fully-connected layers which, as explained in section 4.2, sum up to 87.2% of the total memory taken by the learnable parameters of the network. The next bar shows the size of the parameters file after applying a 80% of neuron-pruning, 153.6 MBytes, which represents a 28% of the original size. Finally, the rightmost bar shows the memory taken by the parameters of the network when combining FP16 with an 80% of neuron-pruning. In this last case, the network takes only 76.8 MBytes, which represents a 14% of the original size.
Chapter 5

Conclusions

5.1 Acquired knowledge

In this project, I have learned about deep learning and deep neural networks. This has complemented the knowledge I have acquired during the degree, since such specialized topics are not covered. I have also learned how to train and validate a state-of-the-art neural network model. In addition, I have extended my knowledge in computer vision, as the neural network analyzed in this project solves detection, a classic computer vision task that combines location and classification. I have also learned how a deep learning framework works from scratch and until the point of being able to modify and extend the source code with my own functionalities. I have extended my knowledge in C++ and, furthermore, I have learned how to interoperate between Python and C++ using the Boost Python library. I have learned a new mechanism for serializing structured data using Protocol Buffers. And finally, I have learned how to measure the CPU and GPU power consumption in a mobile SoC.

5.2 Project results

In this project, several optimizations have been implemented in order to adapt a deep neural network to a low-power environment. The analysis has been carried out in a particular very deep convolutional neural network, Faster R-CNN, using a concrete low-power environment, the Nvidia Jetson TX1 developer kit. Furthermore, a state-of-the-art deep learning framework, Caffe, has been explored and tailored to run on a mobile environment.

We have verified that running real-time inference on Faster R-CNN in a mobile environment is indeed a very ambitious goal. Nevertheless, we have made considerable progress in that direction, achieving remarkable gains in terms of execution time, energy consumption and memory footprint, while maintaining the same prediction accuracy.

A profiling of the neural network in the Jetson TX1 platform has been carried out before applying any optimization, which has led to two observations: (a) the execution time is dominated by convolutional and fully-connected layers, which take about 88% of the total execution time, and (b) the memory footprint is dominated by the fully-connected layers, which take about 87% of the total memory taken by the learnable
parameters of the neural network. Therefore, these results have led us to focus the optimizations in the convolutional and fully-connected layers.

The optimizations that have been carried out in this project exploit two observations:

1. Neural networks do not require 32-bit floating point precision to achieve high prediction accuracy. The nature of neural networks makes them very tolerant to small approximation errors in the internal calculations.

2. Deep neural networks tend to be heavily oversized, with lots of internal redundancy. Once the network has been trained, removing redundant neurons has little to no impact in the prediction accuracy.

We have implemented two optimizations motivated by the aforementioned observations. In first place, the Caffe deep learning framework has been adapted to run with half-precision floating point data types, which represent real numbers with 16 bits instead of 32. Furthermore, the correctness of the implementation of this optimization has been backed up with unit tests per layer. The results show that (a) half-precision floats provide a speedup in the forward pass of 1.29x on average, (b) they reduce the forward pass energy consumption of the whole SoC by 23.55% on average and (c) they reduce the memory footprint of the forward pass by 50%.

In second place, the Faster R-CNN very deep convolutional neural network has been pruned using state-of-the-art techniques. In particular, we have implemented a neuron-pruning technique based on the average L1-norm of the input weights \[60\]. Moreover, the neural network has been retrained after pruning in order to achieve the same prediction accuracy as the original model. In addition, several levels of neuron-pruning have been analyzed in order to find out how much pruning the neural network is able to handle before starting to lose accuracy. The results show that, with a 50% of pruning in the fully-connected layers, (a) the forward pass experiments a speedup of 1.22x on average, (b) the forward pass energy consumption of the whole SoC is reduced by 17% on average and (c) the memory footprint of the forward pass is reduced by 44%. With an 80% of pruning, results show that (a) the forward pass experiments a speedup of 1.51x on average, (b) the forward pass energy consumption of the whole SoC is reduced by 26% on average and (c) the memory footprint of the forward pass is reduced by 72%.

These two optimizations are orthogonal and can be combined to further improve the performance and energy efficiency of the forward pass of a very deep convolutional neural network on a mobile GPU. Our numbers show that (a) the combination of all optimizations provides a speedup of 1.55x on average, (b) the forward pass energy consumption of the whole SoC is reduced by 31% and (c) the memory footprint of the forward pass is reduced by 86%.

To sum up, the realization of this project has allowed to conclude that very deep neural networks can be adapted and optimized to run natively on low-power environments, without the need of an internet connection to leverage computation to cloud servers. Although real-time inference has not been fully achieved, the analysis of the optimizations explored in this project has brought to light how other techniques could be applied to further improve the performance of deep neural networks on mobile environments, so there is still room for improvement.

CHAPTER 5. CONCLUSIONS
Chapter 6

Future work

Several aspects of the project can be extended in order to further improve the performance of deep neural networks on low-power environments and achieve real-time inference in the Faster R-CNN network.

During the development of the project, several improvements have been appreciated with the purpose of further exploiting limited-precision arithmetic (see section 4.3):

- Update the cuDNN library version and adapt the Caffe framework source code to work with Nvidia’s half2 data type. This would increment the FP16 units throughput and provide a 2x performance improvement in the convolutional layers forward pass. After our optimizations on fully-connected layers, now the bottleneck has been moved to the convolutional layers, representing 72.4% of the total execution time in the optimized model.

- Use lower-precision data representations. As stated before, CNNs have proven to be very tolerant to approximation errors in the internal calculations. Also, the large dynamic range of floating point values may be unnecessary, and a fixed-point representation might suffice. Therefore, representations like 8-bit fixed-point can be evaluated.

- Implement per-layer precision selection. Using mixed-precision data types to tailor the accuracy to the requirements of each layer would reduce the memory footprint and the energy consumption of the forward pass.

On the other hand, there exist other pruning techniques that have not been covered in this project (see section 4.4). These techniques could further reduce the total number of parameters of the neural network model, improving the performance and energy efficiency of the forward pass:

- Implement and evaluate other neuron-pruning techniques to achieve an even larger pruning rate without accuracy loss. In [60], two other heuristics are proposed to evaluate the importance of each hidden neuron, based on the entropy of the outputs and the output-weights norm.
• Implement and evaluate weight-pruning and deal with sparsity. In [18], a weight-pruning technique is proposed, which is based on the assumption that near-zero weights are not important. As explained in 4.4.1, removing individual weights produces sparse matrices, so the Caffe framework needs to be adapted to use cuSPARSE instead of cuBLAS for the linear algebra primitives.

• Explore layer-pruning. As explained before, deep neural networks are heavily oversized. This could be extended to the point where full layers are not necessary for achieving the same accuracy. In [62], it is demonstrated that shallow neural nets can be trained to achieve performances equivalent to deeper models. This suggests that some layers could be removed from deep neural nets and still achieve the same accuracy after retraining.

• Implement weight-clustering for network compression. In [18], a weight-sharing technique is proposed, which consists in clustering the weights of a layer and compute the centroid of each cluster. Thus, for each weight only a small index into a table of shared weights needs to be stored. Here, the compression is based on the fact that the indices are stored with less bits than the weights.
Bibliography


BIBLIOGRAPHY 83


Appendices
Appendix A

Faster R-CNN topology
Appendix B

Execution time per layer

<table>
<thead>
<tr>
<th>Layer</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>0.027765036</td>
</tr>
<tr>
<td>relu1_1</td>
<td>0.013719082</td>
</tr>
<tr>
<td>conv1_2</td>
<td>0.112254143</td>
</tr>
<tr>
<td>relu1_2</td>
<td>0.014029980</td>
</tr>
<tr>
<td>pool1</td>
<td>0.010893106</td>
</tr>
<tr>
<td>conv2_1</td>
<td>0.053540945</td>
</tr>
<tr>
<td>relu2_1</td>
<td>0.007370949</td>
</tr>
<tr>
<td>conv2_2</td>
<td>0.093758686</td>
</tr>
<tr>
<td>relu2_2</td>
<td>0.007397175</td>
</tr>
<tr>
<td>pool2</td>
<td>0.005885839</td>
</tr>
<tr>
<td>conv3_1</td>
<td>0.047122955</td>
</tr>
<tr>
<td>relu3_1</td>
<td>0.003822088</td>
</tr>
<tr>
<td>conv3_2</td>
<td>0.088368893</td>
</tr>
<tr>
<td>relu3_2</td>
<td>0.003781796</td>
</tr>
<tr>
<td>conv3_3</td>
<td>0.086972952</td>
</tr>
<tr>
<td>relu3_3</td>
<td>0.003788948</td>
</tr>
<tr>
<td>pool3</td>
<td>0.003062963</td>
</tr>
<tr>
<td>conv4_1</td>
<td>0.043642998</td>
</tr>
<tr>
<td>relu4_1</td>
<td>0.001847029</td>
</tr>
<tr>
<td>conv4_2</td>
<td>0.087609053</td>
</tr>
<tr>
<td>relu4_2</td>
<td>0.001995087</td>
</tr>
<tr>
<td>conv4_3</td>
<td>0.086991072</td>
</tr>
<tr>
<td>relu4_3</td>
<td>0.002015114</td>
</tr>
<tr>
<td>pool4</td>
<td>0.001635075</td>
</tr>
<tr>
<td>conv5_1</td>
<td>0.023099899</td>
</tr>
<tr>
<td>relu5_1</td>
<td>0.000619888</td>
</tr>
<tr>
<td>conv5_2</td>
<td>0.022845030</td>
</tr>
<tr>
<td>relu5_2</td>
<td>0.000563860</td>
</tr>
<tr>
<td>conv5_3</td>
<td>0.022668839</td>
</tr>
<tr>
<td>relu5_3</td>
<td>0.000542879</td>
</tr>
<tr>
<td>conv5_3_relu5_3_0_split</td>
<td>0.000023842</td>
</tr>
<tr>
<td>rpn_conv/3x3</td>
<td>0.022385120</td>
</tr>
<tr>
<td>Layer</td>
<td>Execution Time</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>rpn_relu/3x3</td>
<td>0.000803947</td>
</tr>
<tr>
<td>rpn/output_rpn_relu/3x3_0_split</td>
<td>0.000051022</td>
</tr>
<tr>
<td>rpn_cls_score</td>
<td>0.000843048</td>
</tr>
<tr>
<td>rpn_bbox_pred</td>
<td>0.001513004</td>
</tr>
<tr>
<td>rpn_cls_score_reshape</td>
<td>0.000041008</td>
</tr>
<tr>
<td>rpn_cls_prob</td>
<td>0.000633955</td>
</tr>
<tr>
<td>rpn_cls_prob_reshape</td>
<td>0.000034094</td>
</tr>
<tr>
<td>proposal</td>
<td>0.072909870</td>
</tr>
<tr>
<td>roi_pool5</td>
<td>0.058206797</td>
</tr>
<tr>
<td>fc6</td>
<td>0.746305943</td>
</tr>
<tr>
<td>relu6</td>
<td>0.000962973</td>
</tr>
<tr>
<td>drop6</td>
<td>0.000136137</td>
</tr>
<tr>
<td>fc7</td>
<td>0.068043947</td>
</tr>
<tr>
<td>relu7</td>
<td>0.000746965</td>
</tr>
<tr>
<td>drop7</td>
<td>0.000043869</td>
</tr>
<tr>
<td>fc7_drop7_0_split</td>
<td>0.00307798</td>
</tr>
<tr>
<td>cls_score</td>
<td>0.004835129</td>
</tr>
<tr>
<td>bbox_pred</td>
<td>0.004922867</td>
</tr>
<tr>
<td>cls_prob</td>
<td>0.001355886</td>
</tr>
</tbody>
</table>

**Table B.1:** Execution time per layer.
## Appendix C

### Parameter size per layer

<table>
<thead>
<tr>
<th>Blob</th>
<th>Weights</th>
<th>Biases</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1_1</td>
<td>1728</td>
<td>64</td>
<td>1792</td>
</tr>
<tr>
<td>conv1_2</td>
<td>36864</td>
<td>64</td>
<td>36928</td>
</tr>
<tr>
<td>conv2_1</td>
<td>73728</td>
<td>128</td>
<td>73856</td>
</tr>
<tr>
<td>conv2_2</td>
<td>147456</td>
<td>128</td>
<td>147584</td>
</tr>
<tr>
<td>conv3_1</td>
<td>294912</td>
<td>256</td>
<td>295168</td>
</tr>
<tr>
<td>conv3_2</td>
<td>589824</td>
<td>256</td>
<td>590080</td>
</tr>
<tr>
<td>conv3_3</td>
<td>589824</td>
<td>256</td>
<td>590080</td>
</tr>
<tr>
<td>conv4_1</td>
<td>1179648</td>
<td>512</td>
<td>1180160</td>
</tr>
<tr>
<td>conv4_2</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>conv4_3</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>conv5_1</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>conv5_2</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>conv5_3</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>fc6</td>
<td>102760448</td>
<td>4096</td>
<td>102764544</td>
</tr>
<tr>
<td>fc7</td>
<td>16777216</td>
<td>4096</td>
<td>16781312</td>
</tr>
<tr>
<td>cls_score</td>
<td>86016</td>
<td>21</td>
<td>86037</td>
</tr>
<tr>
<td>bbox_pred</td>
<td>344064</td>
<td>84</td>
<td>344148</td>
</tr>
<tr>
<td>rpn_conv/3x3</td>
<td>2359296</td>
<td>512</td>
<td>2359808</td>
</tr>
<tr>
<td>rpn_cls_score</td>
<td>9216</td>
<td>18</td>
<td>9234</td>
</tr>
<tr>
<td>rpn_bbox_pred</td>
<td>18432</td>
<td>36</td>
<td>18468</td>
</tr>
</tbody>
</table>

Table C.1: Parameter size per layer.
Appendix D

Input-weights norm neuron-pruning script

```
#!/usr/bin/python
import caffe_pb2 as caffe
import sys
import numpy as np

net_param = caffe.NetParameter()

print 'Reading weights from file...

f = open(sys.argv[1], 'rb')
net_param.ParseFromString(f.read())
f.close()

layers = {}
for layer in net_param.layer:
    if layer.name in ['fc6','fc7','cls_score','bbox_pred']:
        blobs = layer.blobs
        data = np.array(blobs[0].data).reshape(blobs[0].shape.dim)
        if len(blobs) == 1:
            layers[layer.name] = data
        elif len(blobs) == 2:
            bias = np.array(blobs[1].data).reshape(blobs[1].shape.dim)
            layers[layer.name] = data, bias

# pruning percentage
p = 0.8

print 'Pruning fc6 layer...

inorm = np.absolute(layers['fc6'][0]).mean(axis=1)
th = inorm[np.argsort(inorm)[int(p*len(inorm))]]
keep1 = inorm>th
data = layers['fc6'][0]
```
bias = layers['fc6'][1]
layers['fc6'] = data[keep1,:,], bias[keep1]

print 'Pruning fc7 layer...'
inorm = np.abs(layers['fc7'][0]).mean(axis=1)
th = inorm[np.argsort(inorm)[int(p*len(inorm))]]
keep2 = inorm>th
data = layers['fc7'][0][:,keep1]
bias = layers['fc7'][1]
layers['fc7'] = data[:,keep2, :], bias[keep2]

print 'Adapting cls_score layer...'
data = layers['cls_score'][0]
bias = layers['cls_score'][1]
layers['cls_score'] = data[:,keep2, :], bias

print 'Adapting bbox_pred layer...'
data = layers['bbox_pred'][0]
bias = layers['bbox_pred'][1]
layers['bbox_pred'] = data[:,keep2, :], bias

print 'Writing weights to file...'
for layer in net_param.layer:
    if layer.name in ['fc6','fc7','cls_score','bbox_pred']:
        blobs = layer.blobs
        data = layers[layer.name][0]
bias = layers[layer.name][1]
        blobs[0].data[:] = data.flatten()
blobs[0].shape.dim[:] = data.shape
        blobs[1].data[:] = bias.flatten()
blobs[1].shape.dim[:] = bias.shape

f = open(sys.argv[1]+'.pruned','wb')
f.write(net_param.SerializeToString())
f.close()
Appendix E

Plot loss script

```python
#!/usr/bin/python
import re
import sys
import matplotlib.pyplot as plt

log_file = sys.argv[1]
f = open(log_file)
points = []
for line in f.readlines():
    if re.search('Iteration.*loss', line):
        it, loss = re.findall('Iteration (.*?), loss = (.*?\n), line[0]
        points.append((int(it), float(loss)))

it, loss = zip(*points)
fig = plt.figure(log_file)
plt.plot(it, loss)
plt.xlabel('iteration')
plt.ylabel('loss')
plt.show()
```
Appendix F

Regions.h

```c
#ifndef __REGIONS_H
#define __REGIONS_H

#include <sys/time.h>
#include <stdio.h>
#include <cuda.h>

#define START_REGION(region_name) {
    struct timeval tv;
    double tstamp;
    cudaDeviceSynchronize();
    gettimeofday(&tv, NULL);
    tstamp = ((double)tv.tv_sec) + (((double)tv.tv_usec) / 1000000.0);
    printf(">>>>> Start %s %0.8f\n", region_name, tstamp);
}

#define STOP_REGION(region_name) {
    struct timeval tv;
    double tstamp;
    cudaDeviceSynchronize();
    gettimeofday(&tv, NULL);
    tstamp = ((double)tv.tv_sec) + (((double)tv.tv_usec) / 1000000.0);
    printf(">>>>> Stop %s %0.8f\n", region_name, tstamp);
}

#endif // __REGIONS_H
```