Convolutional Neural Network Quantisation for Accelerating Inference in Visual Embedded Systems

Konvolutionellt neuronnät kvantisering för att accelerera inferensen i visuella inbyggda system

ANDREA LEOPARDI
Convolutional Neural Network Quantisation for Accelerating Inference in Visual Embedded Systems

ANDREA LEOPARDI
Abstract

The progress within the field of Deep Learning has enabled the realisation of many Computer Vision tasks with a high level of accuracy. In order to attain such achievements, neural networks have been designed gradually with deeper architectures. While this development approach has lead to outperforming complex tasks, it has also entailed an incremental demand for high computational resources. Oppositely, it has emerged at the same time the need of performing Deep Learning techniques on limited-resource devices, requiring local computing and ability of processing at the edge.

Network reduction represents a feasible approach for addressing this issue and enhancing Deep Learning performances on embedded systems and low-resources devices. In particular, network quantisation is here proposed as a versatile and effective method of network reduction. It allows the approximation of neural networks parameters and processing units, accelerating the execution time with no significant losses in accuracy. Algorithms of network quantisation result complementary to other network reduction techniques and can be applied on top of already-designed models.

The study of network quantisation in this work has to be inserted as part of a project consisting in the development of a visual embedded system within the field of Advanced Driver-Assistance Systems. This project exploits Deep Learning for performing object detection operations in real-time. Furthermore, quantisation is adopted to accelerate the inferring of neural networks specifically on Zynq MPSoC platforms.

Evaluations of different quantisation algorithms employable for this work have led to the selection of a model designed on hardware: we have reproduced such a model outside its native framework and analysed it in respect to other models and on different platforms. The achieved performances prove the validity of quantisation as network reduction technique for an inference acceleration. In particular, quantisation results very effective within embedded systems for its handling of integer values instead of floating points and its suitability for improved hardware designs.
Sammanfattning

Framstegen inom djupinlärning har gjort det möjligt att uppnå många datorvisionsuppgifter med hög noggrannhet. För att uppnå sådana prestationer har neurala nätverk utformats gradvis med djupare strukturer. Även om denna utvecklingsmetod har lett till att överträffa komplexa uppgifter, har det också medförts en ökad efterfrågan på höga beräkningsresurser. Samtidigt har det uppstått ett behov av att utföra djupinlärnings tekniker på begränsade resursenheter, vilket kräver lokal databehandling och förmåga att bearbeta vid kanten.


Studien av nätkvantiserings i detta arbete måste införas som ett led i ett projekt i utveckling av ett visuellt inbyggt system inom avancerad förrassistanssystem. Detta projekt utnyttjar djupinlärning för att utföra objektdetekteringsoperationer i realtid. Vidare antas kvantisering för att påskynda inferringen av neurala nätverk specifikt på Zynq MPSoC plattformar.

Utvärderingar av olika kvantiseringsalgoritmer som är anställbara för detta arbete har lett till valet av en modell som är utformad för hårdvara: Vi har reproducerat en sådan modell utanför sitt inhemska ramverk och analyserat den i förhållande till andra modeller och på olika plattformar. De uppnådda prestationerna visar kvantiteten som en nätverksreduktionsteknik för en ökad acceleration. I synnerhet är kvantiseringsresultaten mycket effektiva inom inbyggda system för hantering av heltal i stället för flytande punkter och dess lämplighet för förbättrade hårdvaruutformningar.
## Contents

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Statement</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Purpose</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Applications, Ethics and Sustainability</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Research Methodology</td>
<td>4</td>
</tr>
<tr>
<td>1.5.1 Methodologies for Scientific Writing</td>
<td>4</td>
</tr>
<tr>
<td>1.5.2 Literature Review</td>
<td>5</td>
</tr>
<tr>
<td>1.5.3 Deep Learning and Convolutional Neural Networks</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Delimitations</td>
<td>9</td>
</tr>
<tr>
<td>1.7 Thesis Outlines</td>
<td>10</td>
</tr>
<tr>
<td><strong>2 Theoretic Background</strong></td>
<td>12</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Deep Learning and Convolutional Neural Networks</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Introduction</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 DNN Training</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3 Convolutional Neural Networks</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Embedded Systems for CNN</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Introduction</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 FPGA optimisations for CNN inference acceleration</td>
<td>22</td>
</tr>
<tr>
<td>2.3.3 Algorithmic Optimisation</td>
<td>23</td>
</tr>
<tr>
<td>2.3.4 Data-path Optimisation</td>
<td>25</td>
</tr>
<tr>
<td>2.3.5 Hardware generation design</td>
<td>26</td>
</tr>
</tbody>
</table>
3 Network Quantisation

3.1 Introduction ........................................... 27
3.2 Quantisation ........................................... 29
  3.2.1 Uniform Affine Quantisation ................. 29
  3.2.2 Uniform Affine Quantisation Variations ..... 31
3.3 Binary and Half-wave Gaussian Quantizations .... 32
  3.3.1 Binary Quantisation ......................... 32
  3.3.2 Half-wave Gaussian Quantizations ......... 34
3.4 Tinier2 ................................................. 37
  3.4.1 A quantised Tiny-Yolo model ............... 37
  3.4.2 The PYNQ framework ....................... 40

4 Analysis and Results ....................................... 42
  4.1 Analysis ............................................. 42
  4.2 Models Comparison .................................. 43
  4.3 Inference Times Comparison .................... 46

5 Conclusions and Future Work ............................. 51
  5.1 Conclusions ......................................... 51
  5.2 Future Work ........................................ 52

6 Summary .................................................. 53
List of Figures

1.1 Yolo architecture .................. 8
2.1 Training and Inference computations ................. 13
2.2 Neuron representation ................. 15
2.3 Neural network and DNN representations ........ 16
2.4 Variations of the neural network model ........... 17
3.1 Zynq APSoC architecture ................. 38
3.2 PYNQ framework .................. 41
4.1 Comparison of the inference time on multiple platforms 47
4.2 Detection results .................. 50
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Tiny-Yolov2 architecture</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Tinier2 architecture</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Platforms specifications</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of Tinier2 with canonical models</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>PS and PL performances comparison</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Full-software implementation models on the PYNQ-Z1 platform comparison</td>
<td>49</td>
</tr>
</tbody>
</table>
Acronyms

**ADAS**  Advanced Driver-Assistance Systems.

**AI**  Artificial Intelligence.

**ALU**  Arithmetic-Logic Unit.

**APSoC**  Xilinx Zynq All Programmable SoC.

**ASIC**  Application Specific Integrated Circuit.

**AV**  Automated Vehicle.

**BFLOPS**  Billion FLOPS.

**BNN**  Binary Neural Network.

**CLB**  Configurable Logic Block.

**CNN**  Convolutional Neural Network.

**CPU**  Central Processing Unit.

**CV**  Computer Vision.

**DL**  Deep Learning.

**DNN**  Deep Neural Network.

**DPM**  Deformable Part Model.

**DRAM**  Dynamic Random Access Memory.

**DSE**  Design Space Exploration.

**DSL**  Domain-Specific Language.
DSP  Digital Signal Processing.

FC  Fully-Connected.

FFT  Fast Fourier Transform.

FIFO  First-In First-Out.

FPGA  Field-Programmable Gate Array.

FPS  Frames per Second.

GEMM  General Matrix Multiplication.

GPU  Graphics Processing Unit.

HDL  Hardware-Description Language.

HLS  High-Level Synthesis.

HNM  Hard Negative Mining.

HVS  Human Visual System.

HWGQ  Half-wave Gaussian Quantization.

ICT  Information and Communication Technology.

IoU  Intersection over Union.

LiDAR  Light Detection and Ranging.

LRN  Local Response Normalisation.

LSTM  Long Short-Term Memory network.

LUT  Lookup Table.

MAC  Multiply-Accumulate.

ML  Machine Learning.

MLP  Multi-Layer Perceptrons.

MoC  Model of Computation.
**MPSoc**  Multiprocessor System-on-Chip.

**MSE**  Mean Square Error.

**NMS**  Non-Maximum Suppression.

**OLA**  Overlap-Add Method.

**PE**  Processing Element.

**PL**  Programmable Logic.

**PS**  Processing System.

**R-CNN**  Regional-CNN.

**RNN**  Recurrent Neural Network.

**RTL**  Register-Transfer Level.

**SIMD**  Single Instruction, Multiple Data.

**SIMT**  Single Instruction, Multiple Threads.

**SSD**  Single Shot MultiBox Detector.

**SVM**  Support Vector Machine.

**YOLO**  You Only Look Once.
Chapter 1

Introduction

1.1 Introduction

Presently, images get a central role in our habits and have become massively adopted by technology in many areas of application. Advancements in electronics and data science have enabled machines to "watch" and extrapolate semantics from visual data analogously to human beings. A considerable enhancement has been determined by Deep Learning (DL). This field of study has risen the efficiency in addressing many Computer Vision (CV) tasks, as object detection.

The progresses obtained have allowed the formulation of innovative applications. A considerable driver in this sense is represented by the automotive industry. The capabilities offered by CV in this field are introducing cutting-edge techniques enabling higher standards of security on the road with active safety systems i.e. Advanced Driver-Assistance Systems (ADASs) and leading towards Automated Vehicles (AVs). Different technologies are entailed in these developments, primarily with the aim of analyse the environment surrounding a vehicle. In this perspective, camera have started being adopted in recent vehicles: while their use has already enabled basic applications, such as parking assistance, camera reveal to be a primary means for more advanced propositions, alone or jointly with complementary technologies [1, 2, 3].

While the computation of such techniques has been handled profitably by the mean of dominant cloud systems that rely on capable resources...
as Graphics Processing Units (GPUs), a primary challenge today is focused on real-time computing. Despite a limitation of resources, real-time constraint demands a local processing performed at the edge and not more on the cloud. Embedded systems, and in particular Multi-processor System-on-Chip (MPSoC) devices, can enabled such computation, offering higher or comparable performances in terms of time for execution, energetic efficiency and precision of results. However, since neural networks are usually developed for GPU systems, the adaptation of the design for either or both the embedded devices and the neural networks, is required [4, 5, 6, 7].

We have conducted studies to enable Field-Programmable Gate Array (FPGA) systems enhance object detection techniques within road scenario. The work has been handled at BitSim AB [8] and addressed the development of a real-time embedded vision system for vehicle recognition and localisation and the acceleration of Convolutional Neural Networks (CNNs) on MPSoC environment.

1.2 Problem Statement

Real-time performances require high computational resources. Elevated computing capability, in particular for what it concerns DL are commonly handled by cloud computing resources, which consist in packages of on-demand IT services for custom applications e.g. AWS [9]. If the application is tied to the device at the edge i.e. close to the sensors acquiring the information to analyse, the drawback of a system based on cloud computing is the bottle-neck imposed by the connection between device and cloud. Therefore, it becomes essential to being able to perform complex computation locally at the edge. However, such a task can be challenge because of the computing and energetic limitations imposed by the device itself. Despite this basic assumption, current FPGAs and, more specifically, MPSoC devices support high computation demands through advancements of electronics and of specific designs; this is examined in depth in Section 2.3. Hence the problem to formulate from an Information and Communication Technology (ICT) perspective is how increasing the efficiency of the application throughput, not by improving the electronics design but by remodelling and enhancing the algorithms and techniques
to perform.

Within ADAS, in order to recognise and locate surrounding vehicles, an operation of object detection has to be performed. DL can boost such a task by the means of CNNs. Therefore, the aim of the research is, accordingly with what stated above in this section, accelerating CNNs in embedded environments. For the sake of precision, eventually the objective becomes the study of how to accelerate the inference of a CNN while preserving a decent level of accuracy.

1.3 Purpose

The thesis describes the techniques adopted to accelerate the inferring of CNNs. It provides an overall view of methods related to both the areas of ICT and embedded systems; however, while the latter is outlined in Section 2.3 with no aim for an in-depth analysis, the thesis rests principally on the former. The reader should expects an analysis based on the techniques employed accordingly to the problem stated. Specifically, the work focuses on network quantisation. Quantisation is approached as technique of network reduction and hence it is studied also in correlation with other techniques and methodologies. In this sense, examples of CNNs exploiting algorithms exploring techniques for accelerating inference are considered. Moreover, the analysis is handled firstly in a theoretical manner to be successively tested by implementations. Therefore, a comparison with well-known and affirmed network models gets possible. The work instead does not take into consideration the development of the entire application system, but outlines it briefly in Section 1.6. The decision has been taken to favour the study of a methodology that enables a faster inferring of neural networks, in relation to an academic perspective and to the personal field of study.

To sum up, the thesis focuses on CNN quantisation, handling an analysis on the method itself and in relation to other complementary techniques, from both a theoretical and a practical study.
1.4 Applications, Ethics and Sustainability

A real-time inferring of a neural network enables advanced analyses of the environment surrounding a vehicle. The principal aim is being able to locate and recognise i.e. to detect, road-actors such as pedestrians, cyclists, cars and trucks through a single camera only. The development of this type of vision system can be largely employed in daily-life applications within the automotive field. It allows to increase the security of a vehicle by such an active safety system [3]. In comparison with other technologies, as Light Detection and Ranging (LiDAR) and radar, cameras have already been largely installed on new vehicles to enable both basic and advances applications e.g. parking assistance, surrounding views, adaptive cruise control, etc. Moreover, despite their clear drawback of high sensitivity to luminosity and environmental conditions, they do not depend from the "object-to-detect" material and result being much affordable. Furthermore, rightly because of their technical feasibility and costs, even when other sensors are needed, camera are still deployed as complement [10]. Finally, the development and deployment of affordable road-safety applications enabled by cameras can contribute towards the reduction of the large amount of road-injuries striking low-income countries with a lack of vehicle and infrastructure safety requirements [11].

1.5 Research Methodology

1.5.1 Methodologies for Scientific Writing

The field of [DL] and in particular for what concerns [CNNs] has recently become a trend topic within electronics engineering and computer science. The effectiveness of its results and the robustness of its theoretical background as well as the fervour raised in the community and the crescent number of specific frameworks and resources, have risen the research around [DL] and provided a large amount of studies [12, 13, 14, 15, 16, 17, 18]. Furthermore, with the disclosure of different open-challenges [19, 20, 21, 22] developers have boosted the number of neural network’s proposals and quantitative research around this topic, thus carrying an effective comparison out may result tricky [23]. Therefore, this work is mainly based on a qualitative research rested on
a realistic philosophical assumption. While, accordingly to this choice, the research relies on analytical and applied methodologies that, from the analysis and comparison of methodologies addressing CNN acceleration, validate hypothesis and assess new propositions, it then follows abductive approaches that enable the study of improved and original techniques to address the problem. In this way, it is possible to build a case study research strategy that takes into account and merges both the quantitative and qualitative evidences gathered during the study.

1.5.2 Literature Review

The object detection problem is here defined as the task of classify and locate one or multiple objects in the image where they are represented. Such a task has been addressed with different CV techniques. Two noteworthy approaches are Deformable Part Model (DPM) [24] and Selective Search [25]. Such techniques employ a classifier per object and make evaluations in the image considering different locations and scales. The two methods seek object applying sliding window and region proposal respectively, and from the latter the more advance Regional-CNN (R-CNN) [26] has been proposed; after finding potential bounding boxes by a region proposal approach, R-CNN applies a classification and refine the outcomes by an operation of post-processing respecting the overall results. More recent CNNs are instead simpler and easier to adapt to data. Therefore, modern neural networks represent the main choice for object detection.

Neural networks can be built for different tasks: classification, object detection, segmentation, scene recognition and so on. Since the classification task represent the base for the other ones, many optimisations and part of this work are centred on it. To accelerate the inference phase, the approach undertaken consists in reducing the structure and settings of a "complete" network. Usually such reduced networks are referred to as tiny or small e.g. the reduction of You Only Look Once (YOLO) [27] is referred to as Tiny-Yolo. Because of the way the research is conducted, resources are individually introduced "on-the-way" in Chapters. However, literature review is based on few major resources here presented. For what it concerns notable
models are taken as reference; they are here divided as canonical and reduced or tiny models.

LeNet [28] has been introduced in 1989 and represents a milestone in the history of DL. It was through the first proposals to contribute towards the success of neural network for classification tasks, been employed also for commercial purposes. It was developed with the aim of recognising handwritten digital in grey-scale. It makes use of Fully-Connected (FC) and convolutional layers, average pooling and sigmoid function\(^1\). A successive model that has become a standard is AlexNet [29]. Such a network was proposed in 2012 and is famous to have won the first ImageNet challenge [21]. It increased the number of layers and employed the Local Response Normalisation (LRN), now abandoned for the more common batch normalisation [4]. VGG [30] goes deeper with the number of layers, reaching 4 times the size of LeNet; the size of the filter gets also increased accordingly with the depth of the model. VGG is deployed in 138MB of weights and 15.5G Multiply-Accumulates (MACs) for a 224x224 input image. Later, single serial connections have been replaced by parallel ones by the introduction of the inception module with the GoogLeNet or Inception [31] model. It also map all the weights and activations into GPU to decrease the resources for the training; unitary filters are employed as bottlenecks to reduce the amount of channels per filter [32]. GoogLeNet has been largely adopted and developed in different versions during years [33, 34].

Principally two CNN models have been taken into analysis in this work because of their performances and notability in their field of study: the Single Shot MultiBox Detector (SSD) [35] and the YOLO [27] networks. SSD has been used in the first stage of the overall project to be then replaced with YOLO. However, both the networks have been successful in the community and largely recalled for comparisons or as starting point for further developments [36, 23, 37]. Differently from previous models, such as R-CNN, SSD and YOLO perform localisation and classification in a single evaluation: the image is got in input and passed forwards through the network at once, outcomes all bounding boxes with labels in a single result. Through this approach, the two models achieve fast and accurate performances, reaching real-

\(^1\)These computational techniques are described in Chapter 2
time predictions. Therefore, they have been closely studied as feasible resources for the work.

**SSD** deploys different convolutional layers of different sizes and in particular employs in some parts of the network the so-called extra feature layers composed by 3x3 filters used to estimate the bounding box of an object from the position of the feature maps. An operation of Non-Maximum Suppression (NMS) is applied at the end of the network to refine the result and keep the most confident bounding boxes. Also, Hard Negative Mining (HNM) is applied during the training phase to get rid of negative predictions. **SSD** comes within the 300x300 and 512x512 versions, where the latter deploys an extra convolutional layer for an increased accuracy; nevertheless, the former model can reach a faster inference.

**YOLO** has been developed from the concepts introduced by GoogLeNet and its inception module. It takes in input an image and divides it into a SxS grid: every cell of the grid predicts B bounding boxes and assigns to each of them its relative confidence score each; beside the bounding boxes, the cells divide the image also in regions of prediction. The cells where the centre of an object falls into becomes responsible for its detection. The score assigned to each bounding box depends from both the certainty of effectively containing an object and the accuracy of the box \(^3\); successively the result per cell is given by the confidence of a class \(^4\) for the score of the cell \(^4\). Hence, the resulting tensors quantity depends from the number of cells, bounding boxes and classes \(^5\). As for **SSD**, a NMS \(^2\) is operated at the end of the network. While results are very effective in terms of inference timing and accuracy, **YOLO** proves a light weakness in detecting near objects.

---

\(^2\)True negative results.

\(^3\)The score corresponds to the confidence calculated as:

\[
Pr(Object) \cdot IOU_{pred}^{truth},
\]

where IOU indicates the Intersection Over Union.

\(^4\)class; confidence by score per box:

\[
Pr(Class_i/Object) \cdot Pr(Object) \cdot IOU_{pred}^{truth} = Pr(Class_i \cdot IOU^{truth}_{pred}).
\]

\(^5\)As: \(SxS \times (B \cdot 5 + C)\), where \(SxS\) is the grid, \(B\) the number of bounding boxes per cell, \(5\) the parameters \(i.e. \{(x, y, w, h, \text{confidence})\}\), and \(C\) the number of classes.
1.5.3 Deep Learning and Convolutional Neural Networks

In order to increase the fastness of inference, the approach usually followed is performing a reduction of the number of layers of a model making it tinier. In this sense, YOLO developers have deployed a faster architecture of the model, called Fast-Yolo, and later opened a specific issue for every new version of the network; as mentioned above, these variations are referred to with the name of Tiny-Yolo or just tiny. The notable network addressing inference acceleration are introduced in Chapter 3.

Developments and comparisons of CNNs largely depend from the dataset employed. It is here introduced three canonical dataset largely adopted in this field. MNIST database [38] has been introduced in the nineties and consists in grey-scale 28x28 pixels images of 10 handwritten digits. It is composed by 60000 training plus 10000 test samples. LeNet-5[6] achieved an accuracy of 99.05% and more recent model even more; hence, while it is considered a milestone for datasets, it now used mostly for fast testing. CIFAR-10 [39] has been deployed in 2009 and consists in a dataset of coloured 32x32 pixels images, portraying various objects. It derives from the larger Tiny Image dataset [40]. It

---

recalls 10 mutually exclusive classes with 5000 and 1000 training and test samples each. There is also a CIFAR-100 version containing 100 classes of 500 training and 100 testing samples each; such classes are also grouped into 20 super-classes and hence every sample comes with a fine label referred to the specific class and a coarse one referred to the super-class it belongs to. ImageNet [21] was released in the 2012. It is composed by 1000 classes forming an hierarchy i.e. it recognises synonymous and objects belonging to a same category; for instance it is composed by 120 classes of dogs and allows a classifier to manage the prediction of both the general class "dog" as well as the more specific one. The samples are divided as 1.3M for training, 100k for testing and 50k for validation. ImageNet also introduces the concept of Top-5 and Top-1 accuracy, referred to the maximum accuracy reached respectively by one of the first five categories and to the first one, if correct. As mentioned above, AlexNet was the model to win the first challenge proposed by ImageNet, which is repeated and extended regularly. After the achievement of high accuracy in the solely classification task, object detection was introduced. In 2016 also scene classification and parsing and video samples have been introduced. Other two datasets addressing more complicated tasks than merely classification are Pascal VOC [22] and COCO [20]. The former is composed by 11k of samples for 20 classes and has been formed to handle object detection tasks. The latter, focuses more on contextual information even with a lower amount of categories. In fact, the besides the number of classes and of images, it rise importance the number of instances per image and the precision of localisation information. Larger datasets are being continuously introduced e.g. Open Images [41] by Google and YouTube-8M [42] as large collection of videos.

1.6 Delimitations

The thesis focuses on the acceleration of CNN inference from an academic perspective. Nevertheless, the work has been undertaken as part of an internship at BitSim AB [8] and includes a part concerning the development of a visual system based on MPSoC for a fast recognition of multiple objects within a road environment. Such a system is here briefly described accordingly to the explanations provided in

---

7 Remember ImageNet is structured as a hierarchy.
The system is based on the Xilinx Zynq UltraScale+ MPSoC zcu102 Evaluation Kit \cite{43}. The zcu102 FPGA board has been designed to address automotive, industrial and video and communications applications. From its official description the kit features a Zynq UltraScale+\textsuperscript{TM} MPSoC device with a quad-core ARM\textregistered Cortex-A53, dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit based on Xilinx’s 16nm FinFET+ programmable logic fabric \cite{43}. The system has been built with Vivado SDx tool from the Vivado HLS suite \cite{44}. The system acquires an image in the UYVY format by a wide-angle camera as a stream of data-packages. The frame is formed firstly by the splitting of Luma and Chroma channels into individual streams of data, and successively unified into a frame. Depending from the development issue, the only grey-scale format could been taken into consideration. The frame is then resized and passed in input to a CNN model which gives in output information regarding position and category of the detected objects. For the sake of human-readability, such information are converted into graphical notation onto the original frame, which is then reconverted in streams of data-packages and outputted to an HDMI peripheral. The software has been developed in C/C++ with pragma extensions by Xilinx, and based on the reVISION stack \cite{45}. In particular OpenCV \cite{46}, xfOpenCV \cite{47} and CHaiDNN \cite{48} have been employed for the development.

1.7 Thesis Outlines

The thesis is divided as following:

- Chapter 2 describes the theoretic background that supports the work of the thesis. The chapter is divided into three sections: the first section introduces the concepts behind Artificial Intelligence (AI) and DL leading the reader towards the positioning of neural networks in the field of CV. The second section describes the theory behind neural networks and conducts an excursus regarding all the techniques and methods of CNNs. Through this section, the reader understands the basics behind the architecture that forms a neural network and the approaches employed\footnote{Firstly as xfDNN and successively CHaiDNN\textsuperscript{version 1}.}.
for the part of learning. The third section regards FPGAs and outlines the major methods to accelerate the inference of a CNN by the elaboration of the hardware design. This section takes the distances from the aim of providing an exhaustive explanation of the methods introduced, but instead points out the link between DL and embedded systems.

• Chapter 3 concerns the methodologies for accelerating the inference of CNNs on embedded devices. With this perspective methods of network reduction are investigated with a particular focus to network quantisation. Algorithms of network quantisation are introduced from the basic approaches towards more advanced ones that are eventually adopted for the development of a reduced network employed in the work. Finally, the chapter describes the environment of the FPGA for the implementation and acceleration of the neural network model.

• Chapter 4 presents the analysis conducted to investigate the acceleration provided by the quantisation on embedded systems. Firstly, the architecture of the developed network is compared with canonical models for the evaluation of its level of accuracy. Then, the analysis focuses on the inference time acceleration of the network in relation to the platform and computational resources adopted for the execution.

• Chapter 5 describes the conclusion drawn by the methodologies and analysis. Moreover, it finally outlines future improvements valuable for the continuation of this study.

• Chapter 6 provides a conclusive summary of the work presented in the thesis.
Chapter 2

Theoretic Background

2.1 Introduction

CV is the science and engineering of extracting visual information from raw sensor data. The observation and the study of the Human Visual System (HVS) inspired many CV's tasks and methods, like the object detection problem which became a major issue within the field of CV. The object detection problem, the task of classifying and locating multiple objects within an image, has recently been handled through AI techniques, in particular after the advent of DL. While the natural aim is making a neural network both accurate and fast, currently many efforts start addressing the problem of reducing the time for the inference. For inference is meant the process applied by the network to elaborate new data i.e. data that have not been used in the phase of training, usually at the application-time. This change of direction, which began after the realisation of complex Deep Neural Networks (DNNs) able to reach a decent level of accuracy, is due to the intent of deploying DNNs for devices with limited computational resources and hence open DL to real world applications. In particular, while it is typically required the usage of GPUs as dedicated hardware to achieve fast executions of DNNs because of their computational complexity, now the intent is to maintain the phase of training on the cloud i.e. with GPUs and perform the execution at the edge on embedded environments as FPGAs. Even if GPUs reach state-of-the-art computational performances, FPGAs have been improved and can strike the gap on computation, with the additional asset of offering a higher level of energy efficiency. Moreover, DNNs are being developed incrementing...
sparsity and compacting data types, making FPGAs platforms, designed to facilitate the handling of irregular parallelism and custom data types, preferable to GPUs.

Therefore, inference acceleration can be achieved through either the reduction of DNNs computation costs or dedicated hardware design or a joint solution between the two.

This chapter is organised to firstly introduce in Section 2.1 DNNs and specifically CNNs, giving also an overview regarding their theory, maths and popular models and algorithms, and present in Section 2.3 embedded systems, methods and designs to accelerate CNNs on such platforms.

![Figure 2.1: Training and Inference computations. Illustration from [49].](image)

## 2.2 Deep Learning and Convolutional Neural Networks

### 2.2.1 Introduction

DL can be inserted within the field of the AI and Machine Learning (ML), and more specifically as a specific branch of neural networks; so firstly these latter will be here introduced. The concept of neural networks dates back to the 1940s but the first application appears with LeNet network [28] in the 1989 as automatic learning
for hand-written digit recognition. The idea behind ML is to avoid static and heuristic human-defined programming allowing in a machine a process of artificial learning that is independent from explicit programmer-directives.\footnote{AI approaches have often been stimulated by behaviours of nature and human learning and neural networks in particular can be defined as brain-inspired computation: such a definition derives from the fact that this paradigm of computation takes some of the aspects of human brain; not with the aim of re-create a very similar artificial version of it but just adopting some of its concepts, even because human brain has a complexity that does not allow its full comprehension vanishing the attempt of designing a real emulation.}

Brain-inspired computation adopts the idea of gaining a high-semantic outcome not by a single complex-processing element but through an ensemble of simple and small ones, indeed as brain does with neurons. Human brain is fully populated of interconnected neurons: on average it is formed by 86 billion neurons and up to $10^{15}$ synapses i.e. connections. A neuron is directly connected to dendrites, which bring input signals, and to an axon, which carries the output signal: such signals are referred to as activations; then, the synapse is specifically the joint between an axon and a dendrite. The role of the synapse is to scale the signal received from the axon before transmitting it to the dendrite; this scaling factor is called weight within the brain-inspired computing paradigm, and the process of learning for a network is meant as the setting and adjustment of such weights. A part from neural networks another branch of brain-inspired computation is the spiking computing, which takes into consideration also the width of the signal’s pulse and the relation between timing of different pulses. A representation of the neuron system is given by Figure 2.2.
The output of a neuron is determined calculating firstly the weighted sum of the input values and successively applying a non-linear relation through the activation function; when the sum crosses a certain threshold set by the function, then an output is generated. This computation is defined in Equation 2.2. Different types of activation function are provided in Section 3.3.2.

\[
g(x_i, W_{ij}, b) = \sum_i W_{ij} \times x_i + b; \quad \text{Score function} \quad (2.1)
\]

\[
y_j = f (g(x_i, W_{ij}, b)) = f \left( \sum_i W_{ij} \times x_i + b \right); \quad \text{Activation function} \quad (2.2)
\]

where \(y_j, x_i, W_{ij}, f(\cdot)\) and \(b\) are respectively output and input activations, weights, (non-linear) activation function and bias. \(g(x_i, W, b)\) is the score function.

The output generated by the neuron is then propagated through the network; more precisely, the information is not propagated through a succession of single neurons but rather through layers of neurons. Figure 2.3 represents the schema of a simple neural network: the source information passes-through from the left to the right, being firstly processed to the input layer, which transmits it to one or, usually, several middle layers called hidden layers, which in turn yield the final response given in last instance by the output layer. Neural network
with more than one hidden layer, and hence more than three total layers, are defined as DNNs. DNNs can extract a higher level of features abstraction respect to neural networks and hence are more commonly employed.

Moreover, the previous representation establishes connections directed exclusively from one layer to a consecutive one, making the network a static sequence of operations with no memory; this kind of form models a feed-forward network. Alternatively, a layer with output linked to its input forms a feed-back or recurrent network; in this case, the network has an internal memory for storing the output values and performing intermediate operations, and designs a variant of neural networks called Recurrent Neural Networks (RNNs) and their variant Long Short-Term Memory networks (LSTMs) are more commonly used for speech application cases instead of object detection, and are still based on the weighted sum operation, thus are not treated in this work; moreover, hardware acceleration studies focus more on feed-forward networks than RNNs. Figure 2.4 (a) provides a representation of feed-forward and recurrent network’s layers.
2.2.2 DNN Training

As discussed earlier, ML approaches imply a method of learning: the DNNs learning takes the name of training and is the process of setting the values of the network parameters i.e. the weights’ values; these values are estimated basing on a set of known annotated data-samples provided during this phase. The aim is to create a model able to address a specific task accordingly with the training samples. Successively, the network will use such parameters also in run time with new samples i.e. during the inference phase. Therefore, the model obtained through the training phase should find the best threshold between over-fitting and generalisation; this trade-off depends both from the adopted method of training and from the quality of the provided training data-set.

In other words, given an input the network generates different outputs assigning a score to each of them: the training process should determine the model’s parameters that maximise the score of the correct output and minimise the score of the incorrect ones. The distance between the ideal scores and the actual ones computed by the network is called loss \( L \) and its function can be referred to either as loss function or cost function; the estimation of the loss function can be defined in different ways: the Equations 2.3 and 2.4 report respectively the Multi-class Support Vector Machine (SVM) and the Softmax classifier as examples.
Eventually, the training objective is minimise the average loss. Concretely, the training process performs a process of hill-climbing optimisation to update the weights; this optimization is referred to as gradient descend and updates the weights making use of their relative loss, as exposed in Equation \text{2.5}. In particular, the weight is adjusted in the measure imposed by the leaning rate ($\alpha$): the value of this parameter can largely affect the efficiency and the timing of the training process, so different methods are used to balance it e.g. an approach is to determine the learning rate dynamically according with the loss function. This optimization process is performed iteratively until the convergence of the overall loss function.

$$L_i = \sum_{j \neq y_i} \max (0, s_j - s_{y_i} + \Delta); \quad \text{Multi-class SVM} \quad (2.3)$$

where $s_j = g(x_i, W_{ij}, b)$ is the score function of Equation \text{2.1}, $j$ is the $j$-th class analysed, $y_i$ is the correct class and $\Delta$ is an hyper-parameter representing the margin that the correct class’s score should keep from the incorrect ones.

$$ \begin{cases} L_i = -f_{y_i} + \log \sum_{j} e^{f_j}; & \text{Cross-entropy loss} \\ f_j(z) = \frac{e^{f_j}}{\sum e_{j}}; & \text{Softmax function} \end{cases} \quad (2.4)$$

where $j$ is the $j$-th class analysed, $y_i$ is the correct class and $f_j(z)$ is the softmax function of real-valued scores expressed in $z$. The cross-entropy loss computes the cross-entropy between the true distribution $p$ and the estimated one $q$, as $H(p, q) = -\sum_x p(x)\log (q(x))$; the softmax classifier minimises such cross-entropy.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}}; \quad \text{Gradient descend} \quad (2.5)$$

\text{The performances of SVM and Softmax are comparable and the difference lies more on the lent outcome: SVM provides class scores of more difficult interpretability than Softmax, which gives the the score in terms of probability, but while Softmax loss function pushes the score of the correct class to be high, the SVM loss function drives it to be higher by a fixed margin.}
where $w_{ij}$ is a weight, $t$ the iteration, $\alpha$ the learning rate and $L$ the loss. The gradient in the equation indicates how the specific weight affects the overall model’s function and hence how much it should be adjusted in order to reduce the loss at that iteration.

A convenient way to compute the weights update is propagate backwards through the network i.e. from the output towards the input layer, the error information retrieved by the loss function, and adjust each neuron accordingly to increment the predictive accuracy of the model; this method is called back-propagation. Back-propagation computes the gradients of the loss function with respect to the training data and the network’s parameters applying recursively the chain rule of calculus: since the gradient provides the measure regarding how much a specific input affects the entire function, each weight can be adjusted in relation of the loss function receiving the values of estimated scores and the expected ones gathered from the onwards part of the network. Because of its efficiency in DNN training, back-propagation became largely employed and can be considered the standard algorithm for performing the gradient-based learning in neural networks.

\[
F'(x) = f'(g(x))g'(x) = f'(y)g'(x) = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} = \frac{\partial z}{\partial x}; \quad \text{Chain rule}
\]

the chain rule expressed both in its formal formulation and in the Leibniz’s notation. The two formulations are linked by imposing $z = f(y)$ and $y = g(x)$.

The phase of training can be improved by the use of different techniques: for instance, in order to accelerate and stabilise the training process, each iteration of weights update can be performed on the loss retrieved from multiple input samples rather than from a single one; these data are referred to as batch. Another complementary approach is replacing the random setting of the initial weights using the values of a pre-trained model instead: this technique is called fine-tuning and makes the training process faster and more accurate, besides allowing different methods, as transfer learning, a technique that performs the training on a different dataset.

In general, the training process is performed by a supervised learning
approach, using a set of annotated data-samples. Alternatively, different approaches can be used, as unsupervised learning, based on the clustering of the data-samples, semi-supervised learning, where only a reduced sub-set of the data is annotated, or reinforcement learning, based on a reward-maximisation approach dependent from the actions undertaken by the network within a structured environment.

2.2.3 Convolutional Neural Networks

As seen earlier, DNNs are modelled by a sequence of layers. A layer is formed by multiple weights, each of which is connected to weights from the previous layer and may be connected to weights of the successive one. The set of data passed in input to a layer is known as feature-map. Each layer generates a new feature-map with a higher level of semantics information.

A layer with all the output activations dependent from the weighted sum of all the input ones is defined as a FC layer, or Multi-Layer Perceptrons (MLP); it can be expressed as a matrix multiplication between the inputs and its weights and it is often employed in classification tasks to map the output at the tail of a network to a vector with a value per class. A FC layer entails high computation and storage demands, thus some connection may be set to zero and form a sparsely-connected layer. Figure 2.4(b) represents a FC and a sparsely-connected layers.

To reduce the computational costs of a network another possibility is scaling down the dimensionality of a feature-map by performing a pooling operation, commonly based an average or maximum down-sampling. Pooling is applied separately channel by channel. It contributes to increase robustness and invariance to shifts and distortions.

To improve the training time and the network accuracy, the distribution of the layer input activations can be normalised to get a zero mean and a unit standard deviation. Furthermore, using a batch normalisation technique the covariate shift problem of DNNs [51] the normalised values are scaled and shifted.

It is also possible to limit the amount of weights employed for the output computation making it dependent only from a defined window of
inputs. Furthermore, the use of the same set of weights for the generation of the whole output is referred to as weight sharing. In this fashion, the demand needed to store the network’s weights is reduced. Within CV, it is made a largely adoption of the so called convolutional layer. It consists of a set of filters able to determine discriminative patterns by an operation of convolution with the inputs from the precedent layer. Hence, convolutional layers limit the computation by the use of window and a weight sharing reductions. The behaviour of a convolutional layer is controlled by the stride and padding parameters, which respectively set the shifting unit of convolution and the size of the output e.g. by the use of zero-padding algorithm for instance. DNNs modelled by multiple convolutional layers are referred to as CNNs are the typology of networks considered in this work. Equation 2.7 expresses the computation performed by a convolutional layer.

\[
Y^{\text{conv}}[b, n, u, v] = \beta^{\text{conv}}[n] + \sum_{c=1}^{C} \sum_{j=1}^{J} \sum_{k=1}^{K} X^{\text{conv}}[b, c, v + j, u + k] \cdot \Theta^{\text{conv}}[n, c, j, k],
\]

\[\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, V] \times [1, U];\]

where \(\text{conv}\) is the label to indicate that its element belongs to a convolutional layer, \(\Theta\) is referred to the set of filters, \(X\) to an input volume and \(\beta\) to a bias i.e. an offset applied to the result; \(N, V, U\) represents the size of the feature-map generated from a layer i.e. output feature-map size, while \(B\) is the total number of inputs volumes i.e the batch size, and \(C\) the depth for each of them.

\section*{2.3 Embedded Systems for CNN}

\subsection*{2.3.1 Introduction}

CNNs provide high semantics understanding for many applications regarding images and videos processing, speech recognitions, natural language processing, robotics and so on. Training a network to address a specific task is made only once and it is usually performed on a platform detached from the application edge. This happens to meet
the resources required by the heavy computation and large dataset involved in this stage, like a massive amount of iteration for weights adjustment and a fine calculus precision. Instead, inference can be performed either on the cloud, with powerful resources systems, or at the edge, in the device where sensors are installed: in the former case, the device at the edge has to transmit the acquired information to a cloud server, where the CNN is performed, and then receive the output elaborated; in this case, the transmission efficiency represents the main bottle-neck, in particular for applications involving images. Therefore, the second case, when the CNN is performed at the edge close to the sensor, is more desirable for real-time applications, as ADAS. However, it also entails different challenges, as at first to overcome the constraints of embedded platforms with low-energy consumption and limited memory and computational resources. Therefore, improving the inference of CNNs on embedded systems is of primary importance.

2.3.2 FPGA optimisations for CNN inference acceleration

FPGAs are semiconductor devices based on Configurable Logic Blocks (CLBs) linked by programmable interconnections. Differently from Application Specific Integrated Circuits (ASICs) built with a fixed custom design, FPGAs can be reprogrammed for the desired application. They differentiate from GPU based platforms for their architecture: temporal locality of GPUs can improve the parallelism of operations through techniques that do not allow direct communications between Arithmetic-Logic Units (ALUs) or Processing Elements (PEs), e.g. Single Instruction, Multiple Data (SIMD) or Single Instruction, Multiple Threads (SIMT) execution models; spatial locality of FPGAs models instead the design linking the PEs as a processing chain and enabling them to communicate between each other. Moreover, GPU-based systems exploit a computation based on floating-point for a better calculus precision.

3 A PE is formed by an ALU with local memory. Therefore, a PE is a configurable unit relying on individual computational and caching capabilities, by the use of Digital Signal Processing (DSP) blocs and on-chip registers respectively.

4 Temporal and spatial locality are two basic types of reference locality that define the methodology of memory access.
Being able to compute operations in parallel is crucial for inferring CNNs: convolutional and FC layers are mainly composed of MAC operations\(^5\) that lend themselves to an extensive concurrency, yielding a boost of the network’s timing capability, i.e., increase the speed of the network execution. A MAC operation expects at best two readings and a writing memory accesses entailing a plenty of communications with the memory. Since accessing an external memory, e.g., Dynamic Random Access Memory (DRAM), would imply large periods of latency and a high consumption of energy\(^5\), hardware accelerators can be built ahead of on-chip memories using two levels of caches implemented with a plenty of on-chip buffers for the first level and local register files for the second; this makes memory accesses faster and more energy-effective\(^4\). Moreover, FPGAs boast a parallelism pattern based on an elevated amount of hard-wired DSP blocks reaching up to 20 TMACs\(^6\) that have an ensemble of on-chip memories deployed at nearest for a more efficient memory accessing, as discussed above.

From an hardware point of view, FPGA-based accelerators rely on three main fields of CNN inference optimisation: algorithmic, data-path optimisations and hardware generation designs. These approaches will be introduced and outlined below in this section.

### 2.3.3 Algorithmic Optimisation

Algorithmic Optimisation allows the acceleration of CNNs inference vectorising the implementations of the network’s feature-maps and kernels and reducing the number of arithmetic operations. In order to apply such optimisations it makes use of computational transforms. Computational transforms are largely employed in CPU and GPU systems, but some implementations allow the mapping of CNNs on FPGAs too\(^3\). Three common transforms employed for algorithmic optimisation are the General Matrix Multiplication (GEMM), the Winograd transform and the Fast Fourier Transform (FFT).

---

\(^5\) A MAC unit can be expressed as: \(a \leftarrow a + (b \times c)\).

\(^6\) The equivalent of 8 TFLOPs, against the 12 TFLOPs attained by GPUs\(^4\).
The GEMM is a type of implementation that can be used to concatenate a batch of feature-maps in convolutional and FC layers. With GEMM implementation FC weights can be loaded once per batch, avoiding multiple loads dependent from the number of input to process. A batch of \( B \) elements is modelled as a \( \text{Channels} \cdot \text{Height} \cdot \text{Width} \times B \) matrix \([5]\) and can be used to rise the FC throughput insuring a constant bandwidth of memory demand \([53]\).

Winograd transform \([54]\) is a minimal filter algorithm applied to convolutions with stride equal to one. When the convolution’s kernel is smaller than four the Winograd convolution is performed with the highest efficiency \([55]\). The advantages provided by such an algorithm are reduction of the arithmetic complexity, multiplication-computing of the Winograd transformation matrices \([7]\), implementation of the constants with Lookup Tables (LUTs) and shift-registers \([56]\), and the possibility to make use of loop optimisation techniques, that will be later introduced within data-path optimisations.

The FFT enables the computation of 2D-convolutions as Hadamard product within the frequency domain \([8]\). The profit given by FFT can be employed on FPGA also for the training of a network as made in \([58]\). The arithmetic complexity is thus reduced to \( O(W^2 \log_2(W)) \) and it can be further decreased to \( O(W \log_2(K)) \) by the mean of the Overlap-Add Method (OLA) \([59]\), useful when the size of the signal is much larger than the one for the filter. In general, the use of FFT is wisely applied if the convolution’s kernel size is larger than five \([55]\).

In Table 3 of \([5]\) it is gathered a comparison of the acceleration achieved by these computational transforms.

\[7\] Such transformation matrices are defined within the algorithm \([54]\) and employed in the Winograd filter as it follows:

\[
\begin{align*}
F(u \times u, k \times k) &= C^T \left[ \tilde{\theta} \odot \hat{x} \right] C, \quad \text{Winograd filter} \\
\hat{x} &= A^T x A, \quad x \text{ input feature-map tile of size } (u \times u) \\
\tilde{\theta} &= B^T x B, \quad \theta \text{ filter tile of size } (k \times k)
\end{align*}
\]

where \( A, B, C \) are the transformation matrices and \( \odot \) is the Hadamard product \([4]\).

\[8\] The Hadamard product corresponds to the element-wise matrix multiplication.

\[9\] As it follows \([57]\): \( \text{conv2D}(X[e], \Theta[n, e]) = \text{IFFT} \left( \text{FFT}(X[e]) \odot \text{FFT}(\Theta[n, e]) \right) \).
2.3.4 Data-path Optimisation

Data-path Optimisation aims to take the most of the advantages given by parallelism patterns in CNNs against FPGAs constraints. The techniques allowing to unroll convolutional layers are used also in the GPU environment. SIMD and SIMT execution models introduced in Section 2.3 are an example of acceleration techniques on GPU. Such solutions have been implemented also for embedded devices resulting constrained by memory bandwidth resources: to overcome similar limitations flexible solutions have been proposed. Data are fetched from DRAM to on-chip buffers and processed by PEs; results are then brought back to the past buffers and fetched again to external memories and prepared for the successive layer if needed. It follows that the issue becomes to define an architecture able to optimise the PEs configuration and the temporal scheduling to maximise the computational throughput [5]. Such optimisation can be dealt by the mean of loop optimisation techniques [60]. A Design Space Exploration (DSE) can be performed to evaluate the factors that maximise the computational throughput with the minimum memory access. The Roofline model [61] is adopted for the identification of the most suitable design.

From a different view, the architecture design can be modelled to allow the processing of streams of data by multiple instructions simultaneously; this is described in [62] and is referred to as data-flow Model of Computation (MoC). It is based on the concept that the feed-forward propagation is driven by data and hence its implementation can be bound by memory. The development of a data-flow MoC is built as a network where each node is defined as an actor and abstract data as tokens. An actor consists in a fundamental processing unit that exchanges tokens through First-In First-Out (FIFO) channels with other actors. The execution model is driven by data and hence the execution of an actor, or firing, is caused by the inputs. Such a model, is able to express a CNN where layers’ execution is triggered by the availability in input of a feature-map [61].

2.3.5 Hardware generation design

Hardware generation design defines procedures and instruments for an embedded systems oriented development. More than optimisation techniques it concerns that ensemble of languages and tools suitable to efficiently program an FPGA. Hence, it may be appreciated to mention a few concepts to briefly outline such a topic.

In order to take the most from the advantages offered by FPGAs, it can be fairly apparent the profitability of a Domain-Specific Language (DSL) rather than a generic-purpose approach. DSL-based approaches can improve the inferring of CNN for specific targets. However, beyond this concept it can be effective the use of a higher level of abstraction for electronic systems by the mean of an Hardware-Description Language (HDL). The two most affirmed HDLs are VHDL and Verilog. With these languages it is possible to model and verify electronic circuits and exploit hardware parallelism capabilities. HDLs can be used to create a representation of the circuits through sequential and combinational logic elements which takes the name of Register-Transfer Level (RTL). From RTL it can be derived the design at lower level and thus it is largely employed within this field. It is also used as a mid-step for the FPGA development by High-Level Synthesis (HLS). HLS allows a faster development for more complex applications by the use of higher abstract languages e.g. C/C++, while still exploiting parallelism capabilities of FPGA by pipelines and unrolling optimisations [63]. The Vivado HLS design suite from Xilinx [44] has been employed in this work to implement the application design and accelerate the CNN inference. In Vivado HLS, a C/C++ design is firstly synthesised into RTL design, then simulated and finally implemented for the specific system.
Chapter 3

Network Quantisation

3.1 Introduction

DNNs achieved state-of-the-art characteristics in many field of computer science including CV. Through CNNs it is possible to accomplish with a high precision tasks of classification, object detection and scene description, which bring meaningful advantages in many applications. The progress of these ML techniques has led their development towards large and deep architectures to rise the level of accuracy. However, with such a proposition it has been increased the number of parameters and the depth of architectures comporting higher costs of computation and larger demand of resources. The consequence is that such CNNs are difficult to deploy for limited resources devices. Therefore, a reduction of computational and memory expenses for CNNs is needed. The intended approach aims to implement techniques and methods of network compression for reducing the redundancy of neural networks while preserving a high level of accuracy.

Different techniques have been adopted for network compression. Tensor factorization is a technique that offers a more efficient representation of neural networks reducing the computation costs by the decreasing of the amount of parameters [64, 65, 66]. Pruning is a largely adopted technique. It reduces the architecture’s depth to increase the inference timing at the cost of a lower accuracy. It modifies the network throwing out the redundant connections and parts of a neural network. An example of network pruning is offered by the Tiny version of the YOLO model; however, this techniques is adopted in many
other works [67, 68, 69, 70, 71, 70]. Network quantisation is applied to
the values of network parameters, such as weights and activations, in
order to accelerating the inference, decreasing the computation costs
and saving the storage space [72, 73, 74, 75, 76]. This technique gets
the focus of this work because of its applicability and efficiency; quan-
tisation can be performed on CNNs' architectures under the definition
of certain constraints, chosen in respect of the result to achieve. Fur-
thermore, it can be applied on top of other network compression tech-
niques. Network approximation is a technique very close to quantisation
for its conceptuality. It consists in a low-precision implementation of
neural networks, and in this sense it can be positioned also in a mid-
way between network quantisation and pruning. Even if the method-
ologies undertaken in this work are mainly based on quantisation also
network approximation is considered because of its effectiveness on
inference acceleration and since often models applying quantisation
are also approximated e.g. [77, 78].

In general, quantisation has many advantages: as mentioned, it is
broadly applicable since it is not dependent from specific models and
does not require any particular development. Moreover, despite the
fact that it can be used to convert floating points to integers to more
easily address embedded systems, a quantised network does not re-
quire special hardware platform for being inferred. Quantisation com-
ports a better use of memory and cache, also since a neural network
hardware deployment exploits reuse of parameters that, through quan-
tisation, can be more easily moved and stored, also with a lower power
demand.

Hence, we firstly study quantisation in its general definition and then
we apply it to CNNs parameters: during this procedure, we take into
account the way quantisation affects neural networks’ structures and
functions and therefore how it can be applied accordingly. We also
study the effect of quantisation on canonical models with respect to
accuracy and mostly to inference acceleration.
3.2 Quantisation

Network quantisation on FPGAs mainly addresses how to map floating point values to integers to better suit hardware designs [79]. It follows a brief description of quantisation algorithms that can be used in this sense. Generally, for quantiser it is meant a piecewise constant function which maps all the values within an interval into the same value. It can be defined by Equation 3.1.

\[ Q(x) = q_i, \quad \text{if } x \in (t_i, t_{i+1}], \quad \text{Quantiser} \quad (3.1) \]

with \( q_i \in \mathbb{R}, \forall i \in [1, M] \).

3.2.1 Uniform Affine Quantisation

The mapping for arithmetic on floating point values to arithmetic on 8-bits quantised values has to be affine and follows the relation expressed by Equations 3.2, 3.3.

\[ x_{\text{float}} = A * (x_Q + B), \quad \text{Uniform Affine Mapping} \quad (3.2) \]
\[ x_{\text{float}} = C * x_Q + D, \quad \text{Equivalent form} \quad (3.3) \]

where \( A, B, C \) and \( D \) are some constants and \( x_Q \) the quantised value.

The affine quantisation maps values from a range \((x_{\text{min}}, x_{\text{max}})\) of floating point to one within \(0, N_{\text{levels}} - 1\), where \( N_{\text{levels}} = 256 \) in this case. Quantisation has to take into account that a floating point variable may occur with a value equal to zero. This happens with zero-padding, for instance with input feature-maps in convolutional or pooling layers. Therefore, zero-value has to be exactly represented by the mapping. In order to ensure this, the condition expressed in Equation 3.4 can be considered.

\[ \begin{cases} x_{\text{float}} = 0 \\ x_Q = z \end{cases} \quad (3.2) \Rightarrow 0 = A * (z + B) \Rightarrow B = -z \]
where \( z \) is the zero-point.

Such a condition introduces the zero-point parameter, which assigns a value to the zero. The value of zero-point can be calculated as in Equation 3.5. Notice that the zero-point must be an integer and that, from Equation 3.4, it corresponds to one of the possible quantised values.

\[
\begin{align*}
  x_{\text{float}} &= 0 \\
  x_Q &= z \\
  \Rightarrow 0 &= C \cdot z + D \Rightarrow z = -\frac{B}{A} 
\end{align*}
\] (3.5)

where \( z \) is the zero-point and \(-\frac{B}{A}\) must be an integer.

To complete Equation 3.2, the parameter \( A \) results to be a scale factor, and hence Equation 3.6, which expresses the operation of de-quantising, is completely defined.

\[
x_{\text{float}} = \Delta \cdot (x_Q - z), \quad \text{De-Quantisation} \quad (3.6)
\]

where \( \Delta \) is the scale factor and \( z \) is the zero-point.

Hence, the quantisation is performed as described in Equation 3.7.

\[
\begin{align*}
  x_{\text{int}} &= \text{round}\left(\frac{x_{\text{float}}}{\Delta}\right) + z \\
  x_Q &= \text{clamp}(0, N_{\text{levels}} - 1, x_{\text{int}}) \quad , \quad \text{Uniform Affine Quantisation} \\
\end{align*}
\] (3.7)

where \( \text{clamp}(\text{min}, \text{max}, x) \) is the clamping operator defined as:

\[
\text{clamp}(\text{min}, \text{max}, x) = \begin{cases} 
  \text{min} & x \leq \text{min} \\
  x & \text{min} \leq x \leq \text{max} \\
  \text{max} & x \geq \text{max} 
\end{cases}
\]

The storage of weights and activations is hence reduced to 8-bits of precision. By in-depth optimisations, the uniform affine quantisation can lead to a higher efficiency addressing specific operations \[80\].

Considering a large amount of parameters uniformly distributed, a
uniform quantiser is asymptotically optimal for minimising the mean square quantisation error. This has been proved by [81] and led [74] towards the formalisation of a simple and efficient quantisation scheme which exploits such a property by taking the mean from clusters of network parameters after distributing them uniformly in the feature space. Moreover, [74] has shown as, also within the neural networks’ field, uniform quantisation reaches the best efficiency by the use of variable-length codes.

### 3.2.2 Uniform Affine Quantisation Variations

The uniform affine quantisation can be further simplified imposing the zero-point equal to zero; in such a manner a uniform symmetric quantisation is obtained and the precedent equations are reformulated as described in the system of equations 3.8.

**Uniform Symmetric Quantisation:**

\[
\begin{cases}
    x_{\text{int}} = \text{round}(x_{\text{float}}) \\
    x_Q = \text{clamp}(-\frac{N_{\text{levels}}}{2}, \frac{N_{\text{levels}}}{2} - 1, x_{\text{int}}), \quad \text{with signed int8} \\
    x_Q = \text{clamp}(0, N_{\text{levels}} - 1, x_{\text{int}}), \quad \text{with uint8} \\
    x_{\text{float}} = \Delta * x_Q
\end{cases}
\]

As last, it is here provided a stochastic quantisation. This is added for completeness since in the practice it affects the only rounding operation of the uniform affine quantisation reported in Equation 3.7. This is described in Equation 3.9.

\[
x_{\text{int}} = \text{round}\left(\frac{x_{\text{float}} + \epsilon}{\Delta} + z\right), \quad \text{Stochastic Rounding}
\]

where \(\epsilon\) is a random variable uniformly distributed such as:

\[\epsilon \sim \text{Unif}\left(-\frac{1}{2}, \frac{1}{2}\right)\].

Notice that quantisation and de-quantisation occur as defined by Equations 3.7 and 3.6 respectively. Hence, \(\epsilon\) can be considered as an additive noise and the overall quantisation as a pass-through filter with overbounds saturation [75]. Stochastic quantisation requires specific hardware to be computed during inference which makes difficult its usage, but results valuable during the gradient descend operation in training [75, 82].
3.3 Binary and Half-wave Gaussian Quantizations

The Half-wave Gaussian Quantization (HWGQ) has been recently proposed by [76]. This method partially comes from the binary quantisation approach, so this latter is firstly presented. HWGQ provides an advanced quantisation method able to efficiently decrease the difference of the accuracy between reduced and full-precision networks. Furthermore, HWGQ gets an active part in this work, since it can be employed for adapting neural networks to Xilinx proprietary platforms within the FINN framework, that allows the simulation and real-hardware development of optimised neural networks models [83] on the Xilinx platforms. Since this work has performed some evaluations exploiting such tools, HWGQ is here presented.

3.3.1 Binary Quantisation

The approach adopted by binary quantisation [77] consists in approximating non-linear activation functions exploiting an hyperbolic tangent. The approximation is applied in feed-forward network computation through a piecewise sign function, and in back-propagation during training through a piecewise linear hard hyperbolic tangent function. In consistent classification tasks quantising the activations can entails considerable losses. More specifically, such losses are due to the processing of non-differentiable operator when the method of gradient descend is performed: because of its step-wise responses quantisation might yields to weak gradient signals. At the cost of leading towards a sub-optimal model, such an issue is addressed by the use of continuous approximations during the feed-forward step, that is source of mismatching with the backwards computation [77].

The activation function of Equation 2.2 presented in Section 2.2.1, can be reformulated as in Equation 3.10.

\[ z = g(w^T x), \quad Activation \ Function \]  

(3.10)

---

1 The reformulation aims to use the same notation adopted by [76]. The two expressions are equivalent as demonstrated in [50].
where \( w \in \mathbb{R}^{c \times w \times h} \) is a weight vector, \( x \in \mathbb{R}^{c \times w \times h} \) an input vector, \( g(.) \) a non-linear function and \( c, w, h \) respectively the number of channels, the width and the height of a filter.

Such a representation can be modelled by the means of matrices in CNNs. Weights are then represented by tensors \( W \in \mathbb{R}^{c \times w \times h} \). The approach to quantise this tensor is by the product of a binary matrix with a scale factor, as defined by Equation 3.11.

\[
W \approx \alpha B, \tag{3.11}
\]

with \( \alpha \in \mathbb{R}^+ \) and \( B \in \{1, -1\}^{c \times w \times h} \).

The convolution of an input \( I \) is hence performed as in Equation 3.12 which does not imply multiplication operators and hence reduces the memory usage by a factor of \( \sim 32 \) \[77\]. The optimal approximation through such calculi is achieved imposing the system in Equation 3.13.

\[
I \ast W \approx (I \oplus B)\alpha ; \tag{3.12}
\]

\[
\begin{cases} 
B^* = \text{sign}(W) \\
\alpha^* = \frac{1}{cwh} ||W||_1
\end{cases} \tag{3.13}
\]

where \( \oplus \) indicates a convolution does not exploit any multiplication.

In this manner the convolution can be performed faster with a reasonable approximation on the result, but at the same time it does not address the optimisation of the entire process since the input \( I \) is still represented by floating point values. Therefore, besides the weights also the activation function should be quantised. A first solution is the replacement of the non-linear function \( g(.) \) with the \( \text{sign}(.) \), defined in Equation 3.14 function plus a successive rescale.

\[
\text{sign}(x) = \begin{cases} 
+1, & \text{if } x \geq 0, \\
-1, & \text{otherwise}
\end{cases} \tag{3.14}
\]

However, the benefits of such a solution regard more the only inference and instead yield to a higher complication of the training phase and in particular of the gradient’s computation. \[78\] hence proposed
an approximation of the backwards computation by the means of an \textit{hard hyperbolic tangent} instead of the \textit{sign} function. This way of handling the approximation of the activation function led \cite{77} towards the development of the low-precision XNOR-Net model, and \cite{76} towards \textit{HWGQ} which results more effective also than the supplements proposed by \cite{78}.

### 3.3.2 Half-wave Gaussian Quantizations

The Half-wave Gaussian Quantizations \cite{76} is based on the approximation of the \textit{ReLU} non-linearity, which is an half-wave rectifier largely employed within \textit{DNNs} \cite{84}; it is defined in Equation 3.15.

\[
g(x) = \max(0, x); \quad \text{ReLU} \tag{3.15}
\]

\text{ReLU} is more suitable for back-propagation when used in Equation 3.10 than \textit{3.14} or the hard hyperbolic tangent. Hence \cite{76} propose the approximation of \textit{ReLU} with two different methods to specifically address forward and backward computations respectively. In both the cases a piecewise linear approximation is applied.

Regarding the forward computation, the approximation approach is drastically re-designed. The intuition comes from the consideration of the \textbf{Mean Square Error (MSE)} as measure to define the optimality of a quantiser. Hence an optimal quantiser $Q^*(x)$, respect to the \textbf{MSE}, is defined as in Equation 3.16 and makes the efficiency of quantisation dependent from the statistics of the dot-products of Equation 3.10.

\[
Q^*(x) = \arg\min_{Q} E_x \left[ (Q(x) - x)^2 \right] \tag{3.16}
\]

\[
= \arg\min_{Q} \int p(x) (Q(x) - x)^2 \, dx, \tag{3.17}
\]

\textit{where $p(x)$ is the probability density function of $x$.}

This solution leads to a non-uniform $Q^*$ optimiser and hence a uniformity constrain has to be imposed; successively, the optimal solution

\[^2 q_{i+1} - q_i = \text{const}, \forall i. \text{ The constant value is referred to as quantisation-step.} \]
can be obtained by the means of the iterative Lloyd’s algorithm\cite{85}. This is still a tough solution because at each back-propagation iteration a different quantiser has to be designed per neuron. However, since the dot-product of the activation functions tends to be symmetric with a non-sparse distribution, a Gaussian function can be exploited for its closeness to such statistical structures; this is illustrated in Figure 2 from \cite{76} that shows the dot-product distributions obtained through binary weights and quantized activations on AlexNet. Furthermore, given the half-wave rectification of ReLU, the HWGQ can be eventually proposed. Equation \ref{eq:3.18} gives its definition.

\begin{equation}
Q(x) = \begin{cases} 
q_i, & \text{if } x \in (t_i, t_{i+1}], \\
0, & x \leq 0, 
\end{cases} \quad \text{HWGQ} \tag{3.18}
\end{equation}

where $q_i \in \mathbb{R}^+$, $\forall i \in [1, M]$, and $t_i \in \mathbb{R}^+$, $\forall i \in [1, M + 1]$, with $t_1 = 0$ and $t_{M+1} = \inf$.

Such an algorithm does not prevent from the use of the Lloyd’s algorithm since its parameters vary in dependence from each neuron, but allows to perform it only once with data from the entire network and to benefit from the statistic simplifications provided by the batch normalisation\cite{4}; in fact, the $q^*_i, t^*_i$ parameters do not fairly change across neurons and layers and even through iterations of back-propagation. This gets demonstrated also through empirical results, which shows how under these conditions such distributions can be easily related to Gaussian trends \cite{76}.

The HWGQ is a step-wise constant function which implies that most of its derivatives tend to zero. Hence, the vanishing derivatives problem has to be addressed. As it has been shown above, piecewise linear functions result to be a good solution for weak convergence during the operation of gradient descend; in this case, however, such a function has to approximate both ReLU and HWGQ. \cite{76} proposes three variants of the ReLU function, defined respectively by Equations \ref{eq:3.19}–\ref{eq:3.20}.

\footnote{Which \cite{76} wants to remark it is similar to k-means algorithm.}

\footnote{Batch normalisation, as seen in Chapter 2, ensures zero-mean and unit-variance for each layer.}
\[ \hat{Q}_c(x) = \begin{cases} 
q_m, & x > q_m, \\
q_m + \log(x - \tau), & x \in (0, q_m], \\
0, & \text{otherwise} 
\end{cases} \quad \text{Log-tailed ReLU} \quad (3.20) \]

The ReLU function of Equation 3.15 can be used also in the backwards computation since it is linear piece-wise. However, approximating its derivative in the gradient operation originates a mismatch with the distinct forward one. This generates an inaccuracy in the gradients of the tail of the dot-products distribution making the learning algorithm unstable. The clipped-ReLU function is hence introduced: it differentiates from the previous vanilla version for providing a constant response also with input values that exceed the highest quantisation level \( q_M \). This guarantees zero derivatives within the \((q_M, \infty)\) interval and no mismatches on the tail of the distribution. This provides better results than the vanilla version. However, throwing out the values finishing into the clipped interval the clipped-ReLU comports that with the increment of the number of layers it will be not necessarily implied a correspondent increasing in the accuracy of the network\(^5\). This effect can be alleviated by the introduction of the log-tailed ReLU that enforces a logarithmic growth on the tail of the distribution, behaving like the vanilla ReLU within the quantised interval of values, and weighting down to zero the derivatives computed during the gradient descend operation for the values within the clipped interval [76]. A detailed comparison between low-HWGQ and full-precision models can be found in Table 5 of [76], which substantially shows a reduction of the 10% and 5% on average of the Top-1 and Top-5 accuracy respectively.

The approximation through quantisation provided by HWGQ increases the performances of low-precision networks closing them to their full-precision form and making them suitable for real-time applications.

\(^5\)Considering a number of quantisation levels which tends to the infinite, then the performances should tend to the accuracy’s value of the full-precision model.
3.4 Tinier2

3.4.1 A quantised Tiny-Yolo model

During its entire evolution the work has targeted the Zynq UltraScale+ platforms. While the visual system of Section 1.6 has been developed to specifically address the Xilinx Zynq UltraScale+ MPSoC zcu102 board [43], the analysis regarding quantisation mostly relied on the Xilinx Zynq XC7Z0201CLG400C or, more simply, the PYNQ-Z1 [86]. The PYNQ-Z1 belongs to the Xilinx Zynq All Programmable SoCs (AP-SoCs) family and is designed to feature the open-source PYNQ framework; such a framework has been conceived to enhance the split of development between application, software and hardware levels. Hence it is built on a Linux system that puts into communication basic hardware libraries with a higher and more abstract programming through a Python wrapper interface. Within the PYNQ environment, it has been deployed the FINN framework [83], which has been developed on top of the HWGQ quantisations [77, 76]. In particular, the first model of neural network built with FINN is the Binary Neural Network (BNN) for classification tasks, performing on the MNIST dataset [38]. Beside the BNN, it has been released a YOLO-based model for the PYNQ environment, consisting in a variation of the Yolov2-tiny-voc architecture. In order to perform it on the board achieving high performances, the Tiny YOLO was subjected to some modifications and more specifically to the addition of an extra convolutional layer and to the quantisation of weights and activations with 1 and 3 bits respectively. It is referred to such a variation as Tinier-yolo.

\[6\] It is possible that different networks took the same name of Tinier-Yolo. Here it is referred to the one developed under the QNN-MO-PYNQ project [87]. However, the model was firstly released under another platform with the name of Tiny-YOLO performing at 4.44 Gops/frame with 48.5 mAP of accuracy on the Pascal VOC dataset on the Xilinx Zynq UltraScale+ MPSoC [88].
Therefore, we have decided to re-implement the Tinier-yolo model as a platform-independent network, naming it as Tinier2. The main advantage was the ability to infer this quantised model both on specific FPGA-accelerated design, exploiting all the benefits provided by hardware optimisations, and on every other platform. Thus, we could conduct coherent and compatible analyses on the state-of-the-art quantisation techniques through all CPU, GPU and FPGA devices. We have developed Tinier2 using the Darknet framework as groundwork to remain as much consistent with the original YOLO environment.
For the same reason, we have also trained Tinier2 on the Pascal VOC dataset, fine-tuning from the weights of the YOLO classificator already pre-trained on the Image-Net dataset. In Tables 3.1 and 3.2 the structures of Yolov2-tiny-voc and Tinier2 are provided.

<table>
<thead>
<tr>
<th>layer</th>
<th>filters</th>
<th>size</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>conv</td>
<td>16</td>
<td>3 x 3 / 1</td>
<td>416 x 416 x 3</td>
</tr>
<tr>
<td>1</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>416 x 416 x 16</td>
<td>208 x 208 x 16</td>
</tr>
<tr>
<td>2</td>
<td>conv</td>
<td>32</td>
<td>3 x 3 / 1</td>
<td>208 x 208 x 16</td>
</tr>
<tr>
<td>3</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 32</td>
<td>104 x 104 x 32</td>
</tr>
<tr>
<td>4</td>
<td>conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 32</td>
</tr>
<tr>
<td>5</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 64</td>
<td>52 x 52 x 64</td>
</tr>
<tr>
<td>6</td>
<td>conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 64</td>
</tr>
<tr>
<td>7</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 128</td>
<td>26 x 26 x 128</td>
</tr>
<tr>
<td>8</td>
<td>conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 128</td>
</tr>
<tr>
<td>9</td>
<td>max</td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 256</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>10</td>
<td>conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>11</td>
<td>max</td>
<td>2 x 2 / 1</td>
<td>13 x 13 x 512</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>12</td>
<td>conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>13</td>
<td>conv</td>
<td>1024</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>14</td>
<td>conv</td>
<td>125</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 1024</td>
</tr>
<tr>
<td>15</td>
<td>detection</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Architecture of the second version of Tiny-Yolo model [27].

---

### Tinier2

<table>
<thead>
<tr>
<th>layer</th>
<th>filters</th>
<th>size</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>conv</td>
<td>3 x 3</td>
<td>3 x 1</td>
<td>416 x 416 x 3</td>
</tr>
<tr>
<td>1</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 1</td>
<td>416 x 416 x 3</td>
</tr>
<tr>
<td>2</td>
<td>conv</td>
<td>16</td>
<td>3 x 3 / 1</td>
<td>208 x 208 x 3</td>
</tr>
<tr>
<td>3</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 2 / 2</td>
<td>208 x 208 x 16</td>
</tr>
<tr>
<td>4</td>
<td>conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>104 x 104 x 16</td>
</tr>
<tr>
<td>5</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 2 / 2</td>
<td>104 x 104 x 64</td>
</tr>
<tr>
<td>6</td>
<td>conv</td>
<td>64</td>
<td>3 x 3 / 1</td>
<td>52 x 52 x 64</td>
</tr>
<tr>
<td>7</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 2 / 2</td>
<td>52 x 52 x 64</td>
</tr>
<tr>
<td>8</td>
<td>conv</td>
<td>128</td>
<td>3 x 3 / 1</td>
<td>26 x 26 x 64</td>
</tr>
<tr>
<td>9</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 2 / 2</td>
<td>26 x 26 x 128</td>
</tr>
<tr>
<td>10</td>
<td>conv</td>
<td>256</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 128</td>
</tr>
<tr>
<td>11</td>
<td>max</td>
<td>2 x 2</td>
<td>2 x 2 / 1</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>12</td>
<td>conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 256</td>
</tr>
<tr>
<td>13</td>
<td>conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>14</td>
<td>conv</td>
<td>512</td>
<td>3 x 3 / 1</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>15</td>
<td>conv</td>
<td>125</td>
<td>1 x 1 / 1</td>
<td>13 x 13 x 512</td>
</tr>
<tr>
<td>16</td>
<td>detection</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Architecture of Tinier2 model.

#### 3.4.2 The PYNQ framework

The open-source PYNQ framework developed by Xilinx allows the design of embedded applications taking advantage of the capability from both the language and libraries of Python and the programmable logic and microprocessors of Zynq\[89\]. The framework is designed to enable the programming at different levels, from the hardware up to the software and the applications layers. This structure is outlined in Figure 3.2. In such an environment the Python layer works as a wrapper of the hardware-functions developed into the Programmable Logic (PL). Such functions take the name of overlays. Overlays are developed within the Vivado environment and allow the acceleration of software applications and specific hardware designs. An overlay is formed by a bitstream configuration with its Vivado design .tcl files and a relative Python interface. While the first and the last layers of Tinier-Yolo are

---

\[8\] PYNQ, as name, comes from the combination of python plus zynq.
developed on the Processing System (PS), the entire architecture of the model has been designed on the PL exploiting two overlays that provide 1 bit quantisation for the weights and 2 and 3 bits respectively for the activations. Through these overlays the convolutional and max-pool layers are executed with an accelerated hardware design.

Figure 3.2: Diagram representing the PYNQ framework.
Chapter 4

Analysis and Results

4.1 Analysis

The analysis mainly addresses the performances achieved by Tinier-Yolo and Tinier2 in terms of inference time. The accuracy is considered only from a neural network perspective i.e. how much quantisation affects and drops-down the precision of a model. Hence, as long as the quantisation is applied consistently, the analysis on the accuracy does not depend from a specific platform. The accuracy of Tinier2 is evaluated in comparison to similar models considering characteristics such as inference time, accuracy and weight of the model. Once we have positioned our model in respect to the ones present in literature, the evaluation will focus on its behaviours when performed on different platforms. The evaluations was conducted on CPU, GPU and APSoC platforms; in Table 4.1 it is provided a brief description about the specifics of each. The execution on the PYNQ-Z1 board exploits the accelerated design of Tinier-Yolo; the process involves both the PS with the Cortex-A9 processor of the Zynq board and the PL as further described in Section 4.3. Because of this specific accelerated-hardware design of Tinier-Yolo, the analysis takes into consideration the performances individually achieved by the PS and PL. The reader can find a description of the architecture of Tinier2 network in the precedent section in Table 3.2.
Platform Hardware Specifications

<table>
<thead>
<tr>
<th>Platform</th>
<th>Hardware Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel-Core i7-2670QM at 2.20GHz (4 virtualised units), 11367 MB of RAM on a x64 Ubuntu 16.04.</td>
</tr>
<tr>
<td>GPU</td>
<td>12 EC2 Compute Units (4 virtual cores), 1 NVIDIA K80 (GK210) GPU</td>
</tr>
<tr>
<td>PYNQ (PL)</td>
<td>Programmable logic equivalent to Artix-7 - 13,300 logic slices, with four 6-input LUTs and 8 flip-flops each and 630 KB of fast block RAM</td>
</tr>
<tr>
<td>PYNQ (PS)</td>
<td>650MHz dual-core ARM® Cortex®-A9 processor</td>
</tr>
</tbody>
</table>

Table 4.1: Specifications of the platforms employed in the work.

4.2 Models Comparison

As already mentioned in the previous chapter, we have trained the Tinier2 model on the Pascal VOC dataset [22] for object detection tasks. The accuracy is calculated accordingly to the Pascal VOC original paper in the respect of the Intersection over Union (IoU) between detection and ground-truth and the precision and recall relations. On the same validation set we have estimated the time spent for inference and expressed it in Frames per Second (FPS). The models considered in this comparison offer an effective trade-off between level of accuracy and inference time: while they achieve high precision in the task of detection, at the same time they reach a speed of execution suitable for real-time applications. Table 4.2 presents the results obtained from the evaluation.
<table>
<thead>
<tr>
<th>CR</th>
<th>Model</th>
<th>Train</th>
<th>Test</th>
<th>mAP</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Old YOLO</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>63.4</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>SSD300</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>74.3</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>SSD500</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>76.8</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>76.8</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Yolov2 544x544</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>78.6</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Tiny Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>57.1</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>SSD300</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>41.2</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>SSD500</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>46.5</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Yolov2 608x608</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>48.1</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Tiny Yolov2</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>23.7</td>
<td>244</td>
</tr>
<tr>
<td></td>
<td>YOLOv3-320</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>51.5</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>YOLOv3-416</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>55.3</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>YOLOv3-608</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>57.9</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>YOLOv3-tiny</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>33.1</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>YOLOv3-spp</td>
<td>COCO trainval</td>
<td>test-dev</td>
<td>60.6</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>74.7</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Tiny Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>55.6</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>Tinier2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>40.6</td>
<td>115</td>
</tr>
<tr>
<td>C</td>
<td>Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>74.7</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Tiny Yolov2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>55.6</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>Tinier2</td>
<td>Pascal VOC 2007+2012</td>
<td>2007</td>
<td>40.6</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Table 4.2: With CR it is meant the Computational Resource and platform where the benchmark was performed. The inference time i.e. the FPS may vary with respect to the platform.

- A: NVIDIA GPUs M40 (5.2 cs) and Titan X (6.1 cs).
- B: NVIDIA GPU K80 GPU (3.7 cs).
- C: Intel-Core i7-2670QM, 2.20 GHz (described in Table 4.1).

With cs it is meant the computational capacity score officially assigned by NVIDIA\[90\].

Section A of Table 4.2 reports the performances officially published in [27]. In this section the YOLO model is analysed in its three versions and with all its variations. Also the SSD model is considered in its 300 and 500 deployments. These models were trained on either the Pascal VOC or COCO datasets [22, 20]. It can be noticed that, on average, the accuracy on the Pascal VOC dataset results higher than on
COCO; this is mainly due to the differences between categories and accuracy calculation between the two datasets. The estimations belonging to this section were performed on two GPUs comparable for performances, as specified in the original paper [27] and indicated in the caption of the table. Section B exploits the Tesla K80 Accelerator instead; the K80 features two Tesla GK210 GPUs and a base core clock at 560 MHz. Tinier2 performances are here inserted to allow a wide view of comparison and provide a general understanding about the capabilities of our model. The results are however conditioned by the specific dataset and platform. Therefore, we have performed the same benchmark exploiting the K80 on both the Yolov2 and its tiny-variant models; the choice of these networks was taken basing on the similarity these networks have with Tinier2 for architecture and dataset. Moreover, we have evaluated these three networks also on the CPU platform and shown the results in the Section C of the table.

The evaluation shows that, when compare with Yolov2 and Tiny-Yolov2, Tinier2 has the lowest accuracy but the higher FPS on both GPU and CPU platforms. Tinier2 is respectively 1.36 and 4.1 times faster than YOLO in its tiny and normal versions. Moreover, we can expect that in proportion our model reaches an inference around 279 FPS when performed with the computational resources exploited in Section A. Such a result is in line with expectations: in order to reduce the inference time\(^1\) the accuracy has to drop-down. To reformulate this concept in a more general and intuitive way, one may expect a faster processing when no high precision is required. Going further, the only Tinier2’s architecture counts 2.681 Billion FLOPS (BFLOPS)\(^2\) against the 6.977 and 29.371 of Tiny-Yolov2 and Yolov2, which results less than the half of the operations of the tiny version of yolo. Because of the difficulty in applying quantisation in a fully transparent fashion on non-embedded platforms and frameworks, in this comparison the inference time was evaluated without quantisation, which is instead considered in Section 4.3. Moreover, with pruning and quantisation, the process performed by Tinier-Yolo smooths the feature-maps characteristics and hence the precision of results but leads an effective acceleration of the inference time.

\(^1\)Or increase it, if considered in terms of FPS.
\(^2\)The counted floating-point operations per seconds are meant without quantisation.
4.3 Inference Times Comparison

In order to evaluate the performances achieved by quantisation and network reduction on FPGA, we have carried out an analysis of the inference time spent by Tinier-Yolo and Tinier2 on the different platforms described in Table 4.1. The samples used for the analysis consist in a subset of the Pascal VOC dataset's validation test [22]. While Tinier2 can be performed on all the three platforms, Tinier-Yolo is deeply tied to the PYNQ environment and its accelerated design, and hence its performances have been observed on the Zynq board. Moreover, since the development of Tinier-Yolo involves both PS and PL resources we have estimated the inference time considering the single performance from each computation. The results from the evaluation are presented in Figure 4.1.

Figure 4.1 shows the inference time reached on each platform. In order to provide a basic statistic about the overall performances, the minimum, maximum and mean values of inference time for sample are here reported. Plot (a) presents the performances achieved by Tinier-Yolo on the PYNQ-Z1 platform, considering the comprehensive time from PS and PL, and Tinier2 on the CPU and GPU platforms. Plot (b) refers to the previous one and specifies the performances of Tinier-Yolo discerning between PS and PL computations. Plot (c) refers to the first plot as well and includes into the comparison the inference time of Tinier2 when performed on the PYNQ-Z1 board exploiting the only PS resources. Plot (d) summarises these statistics providing a measure of the efficiency achievable by each platform, expressed as performances on power consumption. Hence, in group A on the left part of the plot, it is provided a representation of the power consumption in terms of Watt and the reachable average inference time in terms of FPS; the latter is depicted in black and inserted on top of the power consumption, which takes instead the colour of the relative platform. Then, in group B on the right, it is shown the efficiency as the ratio of each platform between the previous quantities.

It can be noticed in plot (a) how the performances obtained on GPU outperform the ones on the other platforms; the overall PYNQ inference gets barely faster compared to the one attained on the Intel-Core i7 processor. However, a deeper examination of results, presented in
Figure 4.1: Comparison of the inference time on multiple platforms.
plot (b), shows that considering separately the time spent by the PS and the PL, the performances achieved by the PL are comparable to the ones from GPU. On average, the inference attained by PL is still one order of magnitude higher respect to the GPU’s time, as shown by the values in plot (b), but it results still feasible for real-time applications. Furthermore, considering the performances represented in plot (d) it can be seen that the efficiency achieved by the PL computation of the PYNQ is around 1.6 times higher than the GPU’s one, resulting more suitable for edge-computing applications.

Furthermore, regarding the only PYNQ performances, it is noticeable that more than the 80% of the inference is spent by the PS computation but, on the contrary, the majority of operations is executed on the PL while less than the 5% on the PS. In Table 4.3, the respective performances of PS and PL are compared. Therefore, while this specific implementation of the neural network model bounds the entire inference time to be slowed down by the processing on the PS, the PYNQ framework allows the development of a full hardware acceleration design achieving effective real-time performances.

<table>
<thead>
<tr>
<th></th>
<th>ms</th>
<th>% tot. time</th>
<th>Mops</th>
<th>% tot. ops</th>
<th>Mops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>588.670</td>
<td>80.97 %</td>
<td>171.3254</td>
<td>3.76 %</td>
<td>291.038102</td>
</tr>
<tr>
<td>PL</td>
<td>138.337</td>
<td>19.03 %</td>
<td>4385.9900</td>
<td>96.24 %</td>
<td>31705.111430</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of PS versus PL performances during inference.

This specific benchmark has been released officially within [87].

The only PS computation instead, does not provide suitable performances for real-time applications. However, the Tinier2 model still outperforms the performances reached by comparable models on the same resources. Hence, the quantised model itself provides a higher feasibility for the proposed task. Table 4.4 summaries the compared results. Such a comparison can be considered as a complement of the one presented in Table 4.2; since the accuracy is independent form the platform, the only inference performance is here investigated. The inferring of the models has been performed on the Cortex A9 processor of the PYNQ-Z1; the execution time of our model halves the inference time of Tiny-Yolo. Besides the fact that Tinier2 counts one more
layer than Tiny-Yolo, it can still perform a lower number of operations by reducing the sizes of its convolutional layers. This also makes the Tinier2 model more than two times lighter than Tiny-Yolo, resulting more feasible for cache storage and access.

<table>
<thead>
<tr>
<th>Model</th>
<th>Train</th>
<th>Num. of Layers</th>
<th>Inference [s]</th>
<th>Weights [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yolov2</td>
<td>COCO</td>
<td>30</td>
<td>404.21</td>
<td>194</td>
</tr>
<tr>
<td>Tiny-Yolov2</td>
<td>Pascal VOC</td>
<td>15</td>
<td>75.20</td>
<td>60.5</td>
</tr>
<tr>
<td>Tinier2</td>
<td>Pascal VOC</td>
<td>16</td>
<td>29.10</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 4.4: Full-software implementation models on the PYNQ-Z1 platform comparison.

It follows an extract from the results of the tests executed on both Tinier-Yolo and Tinier2. The examples presented show also cases of imperfect detections in order to provide an idea of the accuracy arisen by the models. The performances however are coherent with the previous analyses.
CHAPTER 4. ANALYSIS AND RESULTS

(a) bird 99%, dog 45%
(b) person 52%
(c) horse 74%, cow 84%, horse 68%
(d) bicycle 78%, dog 86%, car 86%

Figure 4.2: Image examples of the detection results from the models testing. The detected classes with the respective confidence probabilities are specified for each image; the order of appearance is from the left to the right.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

CNNs have seen a rapid growth and development in the recent period becoming feasible for increasingly more complex tasks. In CV CNNs are currently addressing problems of scene recognition after having outperformed the challenges introduced with classification and object detection. However, the need of high-demanding processing on low-resources devices is moving a branch of research in studying methods to accelerate the inferring of neural networks with no considerable loss in accuracy.

Quantisation results a very effective and applicable technique of network reduction that allows to accelerate the speed of network execution and a more efficient cache storage and moving of data within embedded systems. Quantisation can be applied in complement to other techniques and does not require determined specifics from a network architecture.

We have conducted a study on network quantisation techniques for the development of CNN models allowing operations of object detection for real-time tasks on visual embedded systems. In the PYNQ environment we have found a promising framework for addressing the stated problem. This environment offers the right support for the development of the technique and models analysed and offers at the same time tools oriented to hardware design optimisation. In particular we have developed the Tinier2 CNN model basing on the proposed
Tinier-Yolo. Tinier2 releases Tinier-Yolo’s architecture and reduction from the PYNQ environment, allowing the performing on different platforms through a Darknet-based framework. With Tinier2 we have tested the effectiveness of the quantisation assets also with no hardware acceleration support. The network outperforms the inference time of canonical models within all the considered platforms. In particular the PL computation of the Zynq board achieves a speed comparable to the one reached by GPU but an higher efficiency in terms of performances on power consumption. Eventually, Tinier2 enables to operate an accelerated object detection on embedded systems for real-time applications.

5.2 Future Work

The study of this field of neural networks is fascinating and promising for the development of cutting-edge applications. On the short-period, after having realised Tinier2 basing on the Pascal VOC dataset it would be worth for comparison reasons the training on the COCO dataset. Then, in the medium period, it should be handled the implementation of Tinier-Yolo completely on the PL; however, this is also dependent from the work released by Xilinx. Hence, the development of a full-hardware Tinier2 would overcome this limitation. Furthermore, the quantisation approach has been based on the study regarding the HWGQ [76]; its development was enabled by the FINN framework [83] which led to the development of Tinier-Yolo. However, different approaches, as the ones proposed in [82], would have to get included in our Darknet-based groundwork used for the Tinier2 development.

On the long-period these development may be ported to other boards based on the PYNQ environment. The design from scratch of an approximated network would lead to a higher efficiency in addressing specific applications tasks, also by the means of different datasets. In general, network quantisation may be efficiently combined with other methods of network reduction, as made in [77], to can decrease the representation precision minimising the effect on the accuracy.
Chapter 6

Summary

Deep Learning (DL) offers feasible techniques for the accomplishment of complex tasks. The acceleration of Convolutional Neural Networks (CNNs) models to perform operations of object detection with still a high level of accuracy is needed for enabling advanced real-time applications and, in particular, for allowing fast computation on low-resources devices and moving the elaborations from cloud computing resources to local processing at the edge. Being able to achieve this transformation would allow a technological support to many applications such as ADAS.

After the study and the analysis of different techniques for accelerating the inference of CNNs, with a particular focus on quantisation as an effective technique of network reduction, we have proposed the Tinier2 network from the architecture of Tinier-Yolo to release this latter from its specific platform implementation. Tinier-Yolo achieves real-time performances in its PL implementation, reaching comparable times of execution with GPU platform and a higher efficiency. With Tinier2 we have enabled the deployment of advanced techniques of network quantisation on different platforms, outperforming the inference of canonical architectures.

This work proves the assets offered by quantisation in the acceleration of CNNs, in particular when combined with hardware optimisations on embedded platforms.
Bibliography


[17] François Chollet et al. Keras. 2015. URL: https://keras.io.


