POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in Computer Science and Engineering

---

An automated design framework for FPGA-based hardware accelerators of Convolutional Neural Networks

Master thesis of:
Andrea Solazzo
Matr. 836675

Advisor:
Prof. Marco Domenico Santambrogio

Co-advisor:
Ph.D. Gianluca Carlo Durelli

Academic Year 2015/2016

Ringraziamenti

Un primo grazie al mio relatore, Marco, per avermi insegnato molto nel periodo passato in laboratorio, per avermi dato le opportunità per crescere, sia professionalmente che personalmente, e per essere stato un mentore più che un professore, per me e per tutti i ragazzi del *NECST Lab*. Grazie per tutto l'entusiasmo e l'energia che ci metti perché possiamo dare il meglio di noi stessi.

Grazie a Gianluca ed Emanuele, per il loro preziosissimo aiuto nell'intero svolgimento di questa tesi e per avermi dedicato ore preziose per riuscire a portarla a termine, senza di voi non sarebbe stato possibile.
Grazie anche a tutti i ragazzi del laboratorio, per essere sempre pronti a darsi una mano l'un l'altro, e per alleggerire le pesanti giornate di lavoro.

Un Grazie speciale va ai miei compagni di università, Gianmario, Fabio, Matteo, Stefano, Anna, Enrico e tutti gli altri, per tutto il tempo passato insieme in questi anni al Politecnico. Grazie per tutti quei momenti indimenticabili che abbiamo vissuto, tra esami, caffè e un sacco di risate.

Grazie ai miei amici di sempre, Giacomo, Francesca, Andrea, Eleonora e Simone, per essermi stati vicino e per avermi sempre sostenuto. Grazie perché nonostante gli impegni, le distanze e le difficoltà, so che posso sempre contare su di voi, perché nonostante tutto ci siete sempre.

Grazie a mio fratello Stefano, per essere mio complice in tutto. Per essere sempre pronto ad aiutarmi e a sostenermi. Grazie per aver sopportato tutti i cambi di piani improvvisi e tutte le mie assenze. Grazie di essere un fratello eccezionale.
Grazie a Sara, per essere stata sempre presente come amica e compagna di lavoro insieme a Stefano, e grazie anche per il contributo indispensabile alla realizzazione della tesi.

Grazie alla mia compagna di vita, Alessandra, che ha iniziato con me questo percorso. Grazie per essermi stata sempre accanto, per avermi spronato a dare il meglio, per avermi aiutato a risollevarmi nei momenti di difficoltà. Grazie dei tuoi incoraggiamenti e per credere ciecamente in me. Grazie dei tuoi sorrisi e dei tuoi abbracci. Grazie per il tuo amore e per illuminare le mie giornate.

Un ultimo, ma il più importante Grazie va ai miei Genitori, per tutti i sacrifici che hanno fatto per farmi arrivare qui, per tutto il bene che mi hanno sempre dato. Grazie per il vostro sostegno, per aver sopportato tutte le mie risposte brusche, per essere stati sempre pronti ad assecondarmi nelle mie decisioni. Grazie di tutti gli insegnamenti che mi avete dato e di avermi reso la persona che sono. Grazie Mamma e Papà.

Grazie di cuore a tutti voi,
Andrea

Abstract

Convolutional Neural Networks are a particular type of *Artificial Neural Networks* (ANNs) inspired by the biological processes in the primary visual cortex of animals, and represent the state of the art in image recognition and classification. Nowadays, *Convolutional Neural Networks* (CNNs) and other Deep Learning algorithms have been extensively adopted in contexts such as big data analysis and *smart* embedded systems, providing customized technologies through cloud-based services and personalized devices.

As regards this type of applications, the huge amount of data to be processed and power constraints require to find techniques to build fast and energy efficient solutions. In particular, the dataflow pattern of CNN algorithm make them highly suitable for hardware acceleration. In fact, many hardware accelerators have been proposed based on *Graphics Processing Units* (GPUs), *Field-Programmable Gate Arrays* (FPGAs), *Application-Specific Integrated Circuits* (ASICs). Among them, FPGAs are able to make a proper tradeoff between flexibility, performance and power consumption. However, the design and the implementation of a CNN accelerator on such devices may result complex and time consuming, especially for developers that are not experienced in hardware design.

For these reasons, the work presented in this thesis proposes a framework to automatically generate a hardware implementation of CNNs on FPGAs though *High Level Synthesis* (HLS) tools. The working flow of the framework starts from an high level description of the network, integrating TensorFlow for training and an internally developed C++ library for the final implementation.

The proposed approach has been evaluated on several CNN topologies and two different datasets of input images, reducing the design time from hours to minutes, and obtaining hardware implementations able to achieve a *performance-per-watt* ratio up to 10771 $^{FPS}/_{W}$.

Estratto

Le *Reti Neurali Convoluzionali* (conosciute come *Convolutional Neural Networks*) sono un particolare tipo di *Rete Neurale Artificiale*, il cui funzionamento è ispirato a cellule presenti nella corteccia visiva degli animali, e rappresenta oggi la miglior soluzione per il riconoscimento e la classificazione di immagini. Al giorno d'oggi, le cosiddette *Convolutional Neural Networks* (CNNs) e altri tipi di algoirtmi appartenenti alla branca del Deep Learning vengono ampiamente utilizzati in contesti come quelli dell'analisi di big data e dei sistemi embedded *smart*, fornendo delle tecnologie personalizzate attraverso servizi cloud-based e dispositivi come ad esempio smartphones, e smart watches.
In questo tipo di applicazioni, l'enorme mole di dati da processare e i vincoli di consumo energetico rendono cruciale l'individuazione di soluzioni che siano sia veloci che efficienti da un punto di vista energetico. In particolare, lo specifico flusso di calcolo di una CNN rende questo tipo di algoritmi estremamente adatti per essere accelerati in dispositivi hardware dedicati. Infatti, molti acceleratori hardware basati su *Graphics Processing Units* (GPUs), *Field-Programmable Gate Arrays* (FPGAs) ed *Application-Specific Integrated Circuits* (ASICs) sono stati proposti a questo scopo. Tra questi, le FPGA sono in grado di fornire un giusto compromesso tra flessibilità, performance e consumo energetico. Tuttavia, il design e l'implementazione di un acceleratore per una CNN su questo tipo di dispositivi potrebbe risultare sia complesso che oneroso in termini di tempo di sviluppo, specialmente per sviluppatori con poca esperienza di progettazione hardware.
Per questi motivi, il lavoro proposto in questa tesi propone un framework in grado di generare automaticamente un'implementazione hardware di CNN su FPGA attraverso strumenti di *High Level Synthesis* (HLS). Il flusso di lavoro del framework parte da una descrizione ad alto livello della rete, integrandosi con il framework di Machine Learning TensorFlow per l'addestramento e una libreria C++ sviluppata internamente per l'implementazione finale.

La metodologia proposta è stata valutata su diverse topologie di CNN e due diversi dataset di immagini in input, riducendo i tempi di sviluppo da ore a minuti, e ottenendo implementazioni hardware in grado di raggiungere un picco di *performance-per-watt* di 10771 $^{FPS}/_{W}$.

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

ANN	Artificial Neural Network	iv
APSoC	All-Programmable System on Chip	48
ASIC	Application-Specific Integrated Circuit	iv
BRAM	Block-RAM	42
CNN	Convolutional Neural Network	iv
CPU	Central Processing Unit	24
DMA	Direct Memory Access	49
DSP	Digital Signal Processor	47
FIFO	First-In First-Out	40
FLOP	Floating-point Operation	xii
FPGA	Field-Programmable Gate Array	iv
FPS	Frames Per Second	xii
GFLOP	Giga Floating-point Operation	61
GFLOPS	Giga Floating-point Operations per Second	xiii
GPU	Graphics Processing Unit	iv
GUI	Graphical User Interface	6
HDL	Hardware Description Language	5

HLS High Level Synthesis iv

HPC High Performance Computing 3

IoT Internet of Things 2

IR Intermediate Representation 5

LUT Look-Up Table 42

MFLOP Mega Floating-point Operation 61

MIG Memory Interface Generator 49

ML Machine Learning 2

MLP Multi-Layer Perceptron 13

NN Neural Network 2

OCR Optical Character Recognition 23

ReLU Rectified Linear Unit 11

SGD Stochastic Gradient Descent 14

SoC System on Chip 48

SVM Support Vector Machine 2

UI User Interface 5

USPS U.S. Postal Service xii

Introduction 1

This introductive Chapter provides an overview of the topics that will be discussed in this dissertation. Section 1.1 describes the Deep Learning context, a widely discussed topic in both research and engineering. Afterwards, CNNs are introduced as one of the most promising algorithms in this context, briefly describing the idea behind their working principle and providing examples of application fields in which this kind of algorithm are involved. Section 1.2 explains the rationale behind the hardware acceleration of CNNs, focusing on FPGAs as target devices for the implementation of such accelerators. Finally, Section 1.3 describes the purposes of this thesis, providing an overview of the work and its organization.

1.1 The Deep Learning Context

In the last years, the Deep Learning field has emerged as the most promising approach for meeting several challenges in computing. In particular, every day millions of images, text documents, audio file and so on are generated and stored in data-centers of different service providers. Due to this huge amount of data, it is crucial to find techniques to extract from them useful information automatically. This is the so called *Big Data Analysis* field, in which several *Machine Learning* (ML) algorithms, such as *Neural Networks* (NNs), *Support Vector Machines* (SVMs), Random Forests, have been adopted to provide models of the data from an high level of abstraction. Indeed, many datacenter applications currently rely on Machine Learning algorithm to provide services to end users. In fact, prominent examples of use of deep learning algorithms can be found on online available services such as AWS by Amazon [2], Microsoft Bing document analysis [3], and many cloud-assisted services such as Siri [4]) and Google Photos [5].

Moving from *big* to *small* scale, another challenge that has to be faced is the one represented by the ubiquity of smart devices, designed to be always connected and ready to interact with the sorrounding environment. This scenario is the so-called *Internet of Things* (IoT) paradigm, that refers to the interconnection between all kinds of physical objects. IoT devices collect information on and/or interact with the environment, resulting in a plethora of new applications in the domains of smart cities [6], healthcare [7], transportation, and more [8, 9]. These pervasive devices are generally required to understand the environment they are working in and the goals of their target user, possibly improving over time by learning from experience.

Even though the energy efficiency requirement was the most constraining aspect at the beginning of the *mobile era*, use case scenarios and user expectations have changed, and performance steadily gained relevance in the IoT industry. The search for the right trade-off between the two becomes more and more difficult as smart devices (smart watches, smart glasses, and wearables in general) are now entering the market as the next generation personal devices: in this big picture, customization will soon become a key performance indicator.

Deep learning [10, 1] has emerged as the most promising approaches for meeting these challenges. In particular, within the deep learning class of algorithm, CNNs [11] have emerged as the most efficient approach for image recognition and classification tasks.

CNNs are inspired by the biological process in the visual cortex of animals. From an high level of abstraction, they are able to extract features from an image and aggregate this information to provide a classification of the image subject. The effectiveness and performance of such algorithms have considerably improved in the past years, outperforming other existing visual recognition algorithm [1] and becoming the state-of-the-art in image classification.

As a result, in the last years a huge research and engineering effort has been devoted to design, implement and improve these kind of algorithms. Indeed many Machine Learning frameworks have been developed to support tools to build and train CNNs, such as TensorFlow [12], Caffe [13], Torch [14]. The interest in CNNs and neural networks in general lead also to look for specialized and efficient solutions oriented to Deep Learning algorithms, such as the *Tensor Processing Unit* (TPU) announced by Google [15].

1.2 Hardware Acceleration

Despite all the benefits, the adoption of deep learning algorithms, and CNNs in particular, faces significant barriers due to performance and energy requirements at different scales. In the context of *High Performance Computing* (HPC), datacenters have to process data at *Exascale*, consuming a huge amount of power. For this reason, it is crucial to find techniques to speed up the computation, while meeting constraints on power consumption. The same challenges occur also in the IoT field, where the limited resources of mobile devices make often infeasible to implement deep learning algorithms on target hardware [16].

CNNs represent a peculiar type of algorithm where the demands in terms of required memory and number of operations increase exponentially with the size of the network. As an example, in Table 1.1 are reported the number of operations of some well-known networks.

Nevertheless, the specific computation pattern of CNNs make them highly suitable for hardware acceleration. For these reasons, many accelerators have been proposed in the literature based on GPUs, FPGAs and ASICs [18, 19, 20].

Network	FLOPs
LeNet-5 [11]	0.84M
AlexNet [1]	1.4G
GoogleNet [17]	1.5G

Table 1.1: Number of FLOPs of well-known CNNs in the state of the art.

Among these different types of devices for hardware acceleration, FPGAs represent a proper tradeoff between performance and power consumption. On the one hand, the computational pattern of CNNs presents a highly repetitive, pipelined and parallelized structure, which particularly fits with the regular configurable fabric of such devices. On the other hand, even thouhg FPGAs may not reach the same performance peaks of GPUs, their power efficiency makes them suitable for the acceleration of algorithms even in case of strict power constraints.

FPGAs have been succesfully employed as new hardware accelerator platform both in HPC [21] and embedded systems contexts. As a consequence, various vendors, such as Xilinx and Altera (now Intel), have started the production of specific FPGA families for embedded and mobile markets ([22, 23]) able to offer reasonable performance while presenting a low power profile, as well as high-end devices such as the Virtex FPGA [24] and the Stratix-V [25]. Thanks to their reconfigurability, this kind of devices also provide high flexibility, which allows to perform fast prototyping and development round. This is especially needed in the machine learning field, where the design process may require several attempts due to the huge design space and the amount of configurable hyper-parameters, which have to be tuned in order to find the optimal model to accomplish a given task.

However, FPGA design presents a considerable challenge: the learning curve required to target hardware design is indeed very steep. Hence, even an expert software developer might need a considerably long training period to generate a working solution with sufficient performances. This has to be further improved to gather the desired standard. To overcome this issue, HLS can be very helpful, allowing to use high-level languages such as C, C++ and OpenCL to design custom hardware accelerators on FPGAs. Indeed, many industrial tools such as Vivado HLS by Xilinx nowadays are able to obtain efficient solutions comparable with accelerators written with the so-called *Hardware Description Languages* (HDLs) such as VHDL and Verilog. With HLS, it is possible to rapidly explore the design space both from hardware (i.e. the accelerator) and software (e.g. the structure of a CNN) perspectives.

Nevertheless, the benefits introduced by HLS to target FPGAs as devices for hardware acceleration are still not enough when compared to the fast development-round offered by productivity-level languages such as Python or Lua. While a certain level of experience is required to design an efficient hardware accelerator with HLS tools, modern frameworks available online such as TensorFlow [12], Caffe [13], Torch [14], permit to implement several machine learning algorithms using APIs for different languages, significantly reducing the development effort when compared to the actual implementation workflow for FPGA design.

Thus, it is evident the existing gap between modern tools and solid development frameworks in order to target FPGAs.

1.3 Proposed Work and Contributions

The aim of this thesis is to bridge the aformentioned gap between productity-level tools and FPGA-based accelerator design, providing a framework for the automated generation and synthesis of an hardware implementation of CNNs. In particular, the framework allows the user to give an high-level description of the network, specifying the hyperparameters of the different layers. The CNN specification is then translated into an internal *Intermediate Representation* (IR), from which the C++ code implementing the model is generated, along with the *.tcl* scripts to perform the HLS and the bitstream file needed to set up the obtained hardware accelerator on the target FPGA device.

Figure 1.1 shows the overall structure of the framework. The blue modules describe the components implemented in the framework, while the dashed ones represent optional components that may be integrated in the proposed workflow. As shown in the Figure, the developed framework provides a toolchain that connects an high-level *User Interface* (UI) to industrial tools for hardware implementation. Specifically, the structure and the configuration of the network can be

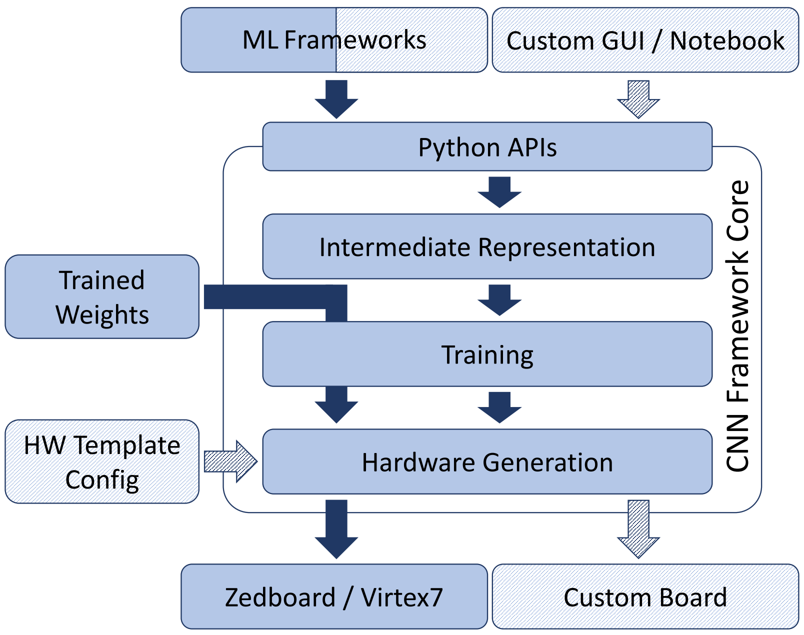

Figure 1.1: Overall structure of the proposed framework.

given through lightweight data-interchange objects such as the ones generated by the Caffe Deep Learning framework by Berkley [13]. Moreover, it is always possible to convert any description generated by other Machine Learning frameworks, and convert it to a compatible representation through the Python APIs exposed by the proposed framework. Thanks to the flexibility of Python APIs, a *Graphical User Interface* (GUI) could be potentially integrated in the work-flow, as well as other types of visual tools such as *Jupyter Notebooks* [26].

Furthermore, the toolchain provides a module to perform the training of the specified CNN and export the weights to the hardware generation flow. Alternatively, a set of weights pretrained in a different environment can be directly included.

As regards the hardware acceleration, the framework include templatized C++ libraries for the High Level Synthesis of the CNN accelerator, which can be used either as a black-box, or configured to meet particular needs for a given application. The hardware generation targets Xilinx devices and it is interfaced with the Vivado Design Suite [27] for the HLS and the bitstream generation. In particular, the currently supported platforms are the Zybo and the Zedboard boards by Digilent [22, 23], powered by a Xilinx Zynq MPSoC and mainly focused on mobile and embedded market, and the Xilinx Virtex7 VC707 Development Board [24], an high-end board oriented to HPC. However, the proposed approach can be easily extended in the future to include any custom board as target device.

In summary, this work includes:

- A novel framework written in Python, providing a set of modules that implement the toolchain for the design and the implementation of CNNs on FPGAs;
- A flexible internal representation based on Google Protocol Buffers that is compliant with a subset of the layer definitions of the Caffe deep learning framework, giving the possibility to provide existing models as input;
- The integration with TensorFlow for CNN training, providing the training set and the test set directly to the framework;
- A hardware library with customizable modules implementing the different type of layers of CNNs.

The proposed approach has been tested on a large experimental set, demonstrating the effectiveness of the proposed solution on well-known case studies as the MNIST [28] and the CIFAR-10 [29] datasets.

The following part of this thesis work is organized as follows. Chapter 2 provides the background on CNNs, giving the main definitions to understand the working principle of such networks. Chapter ?? discusses the literature on both CNN design and hardware acceleration targeting FPGAs. The proposed framework is presented in Chapter 3, discussing the design flow and the details of the modules. Chapter 4 describes the architectural template for the FPGA-based CNN accelerator. Chapter 5 illustrates the experimental evaluation of the proposed approach on different case studies. Finally, Chapter 6 draws the conclusions of the thesis, also including future works.

Background on Convolutional Neural Networks 2

This chapter provides the main definitions needed to understand how Convolutional Neural Networks work. Section 2.1 briefly introduces ANNs in general, explaining their mechanism and the training process. Section 2.4 provides the idea behind CNNs are implemented and the difference with respect to a classical ANN. Section 2.3 gives the details about the overall structure of such networks, discussing the different types of layers that can be part of a CNN. Then, Section 2.4 analyzes research works regarding CNNs in general, focusing on proposed network topologies and application case studies. Finally, Section 2.5 shows the research effort in accelerate such networks on dedicated hardware, such as GPUs, FPGAs and even ASICs.

2.1 Artificial Neural Networks

Back in 1957, a probabilistic model called *Perceptron* was presented by Rosenblatt [30]. This model was firstly inspired by the biological neurons of animal brain. As it can be seen from Figure 2.1, in a simplified way, a neuron is made by the *nucleus*, the *dendrites* and the *axon*. This type of cell is electrically excitable and transmits signals to the other cells through connections called *synapses*, which are formed between the terminal part of the axon and the dendrites of the other neurons. Each neurons receive electrical signals from its dendrites, accumulating charge until a certain threshold is reached. At this point the neuron *fires* the signal through the axon to the other neurons.

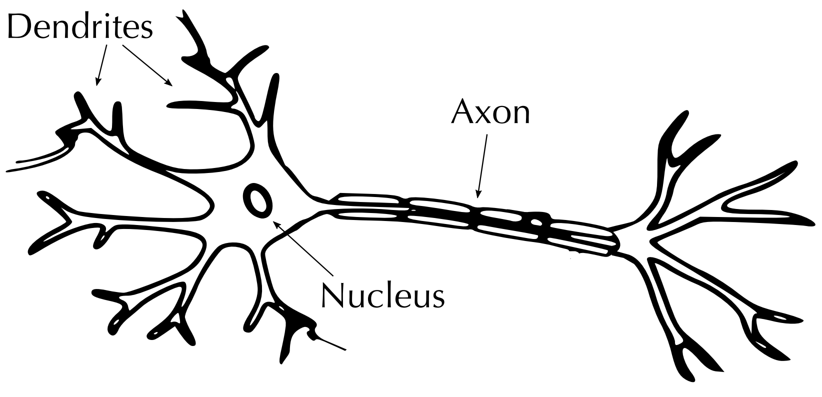

Figure 2.1: The schematic representation of a neuron cell of animals.

2.1.1 The Perceptron classifier

This mechanism is somehow mimicked in the *perceptron* model, represented in Figure 2.2. A perceptron takes an arbitrary number of inputs $x_1, x_2, ..., x_n$ and produces an output y. The connections between the input values and the nucleus, called in this case *synapses*, are *weighted* with values $w_1, w_2, ..., w_n$, while w_0 is a special weight that represents the threshold of the neuron, called *bias*. The nucleus of the perceptron computes a function $f(\cdot)$ of the input, also called *activation function*. The most commonly used activation functions for perceptrons are the *Heaviside step* function, the *sigmoid* and the *hyperbolic tangent* functions, that are smoother than the step function and are real-valued in $[0, 1]$ and $[-1, 1]$

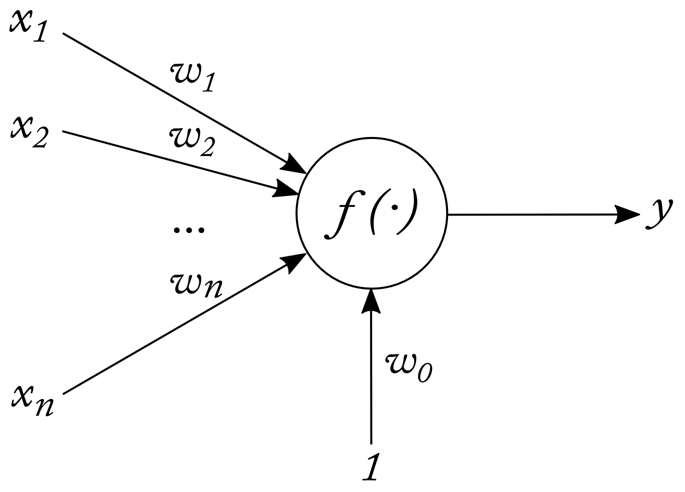

FIGURE 2.2: Diagram of the *Perceptron* model.

respectively, and in some cases the *Rectified Linear Unit* (ReLU) function, defined as the $max(0, x)$.

Given these values, the way in which the output y is computed is really simple:

$$y = f\left(\sum_{i=1}^{n}(w_i \cdot x_i) + b\right) \tag{2.1}$$

Where the bias b is defined as $b = 1 \cdot w_0$. From Equation 2.1, the output of a perceptron is computed as a function of the weighted sum of the inputs. Supposing that the activation function f is the step function, the output of the perceptron would be:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n}(w_i \cdot x_i) + b \geq 0, \\ 0 & \text{if } \sum_{i=1}^{n}(w_i \cdot x_i) + b < 0 \end{cases} \tag{2.2}$$

Drawing a comparison with the neuron cell, the perceptron will *fire* if the weighted sum of the inputs is above a certain threshold (for instance considering $threshold = -b$), otherwise the output would be 0.

Given the right set of weights, the perceptron is able to discriminate any *linearly-separable* set of inputs. However, the great advantage of this model is that the weights can be learned from experience. In other words, it is possible to develop a learning algorithm in order to find the proper values of the weights that allow to obtain the desired value in output.

Consider a set of S samples $\mathcal{D} = \{(x_1, t_1), (x_2, t_2), ..., (x_s, t_s)\}$, where each (x_i, t_i) is a tuple of the n-dimensional vector of the inputs x_i, and the corresponding desired output t_i. This is the so-called *training set*, from which is possible to *learn* the right values of the weights so that $y \to t$. Then, the learning algorithm of the perceptron is executed for a certain number of iterations called *epochs*. The algorithm works as follows:

1. At epoch 0, the weights and the bias are initialized to 0 or to an another pre-determined or random value.
2. For each sample s in the training set, the following steps are performed:
    a. Compute the output y_s with the input vector x_s.
    b. Compute the error with respect to the desired output as:
    $$\delta = t_s - y_s$$
    c. Compute the new values for weights and the bias at step $k+1$ with the following rule:
    $$w_i^{k+1} = w_i^k + \gamma \cdot \delta \cdot x_{s,i} \qquad \forall i \in [1, n] \tag{2.3}$$
    $$b^{k+1} = b^k + \gamma \cdot \delta \tag{2.4}$$
3. Repeat from step 2 until the desired output is obtained.

The parameter γ in 2.3 and 2.4 is the *learning rate*, usually valued in $(0, 1)$, which is able to *tune* the impact of the error δ on the actual weight. After each update during the training process, the value of the weights move towards the correct one. However, it is necessary to find a criterion to obtain good generalization capability, so that the model is able to provide the desired output, even in case of *unseen* inputs. Finding the right number of *epochs* for the training process, as well as the proper value of the *learning rate*, is not an easy task. Indeed, one major problem of this type of learning is the so-called *overfitting*, which occurs when the weights perfectly fit the instrinsic model of the training set, providing garbage results in case of new inputs. For this reason, several techniques have been proposed in order to avoid overfitting and optimize the training results.

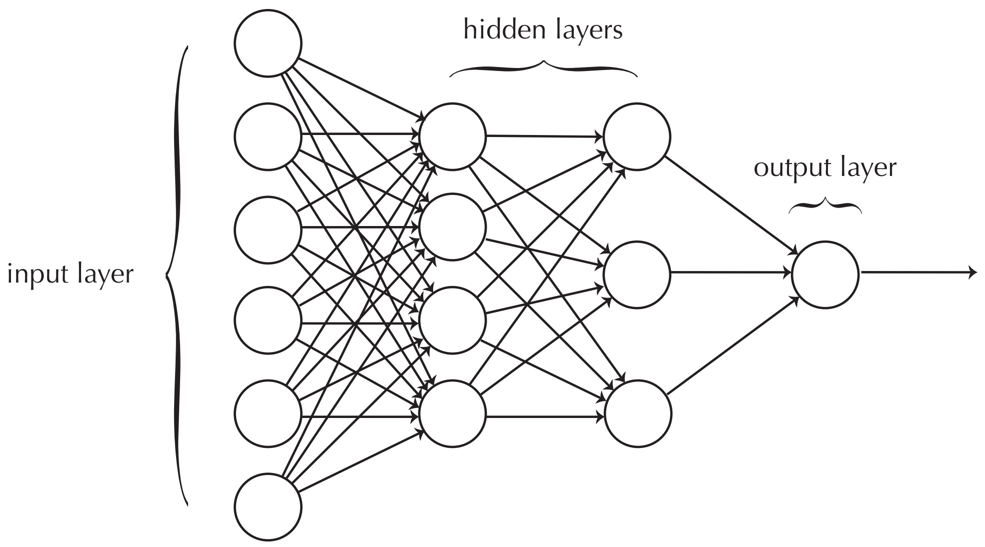

Figure 2.3: Artificial Neural Network topology. Image from *Neural Networks and Deep Learning* book by Nielsen [32].

Although powerful, the perceptron has the strong limitation of being able to discriminate only linearly-separable class of problems. To overcome this issue, the adopted solution was to connect several perceptrons in a network of neurons, analogously to what happens in the animal brain. Even though it may seem a trivial idea, this led to the birth of the well-known machine learning algorithm called *Artificial Neural Network* (ANN), *Neural Network* (NN) or sometimes *Multi-Layer Perceptron* (MLP). As it can be seen from Figure 2.3, neural networks are organized in an arbitrary number of layers of perceptron units. The first and the last layers are called *input* and *output* layers respectively, while the layers in the middle are known as *hidden* layers. Starting from the input layer, the output of the neurons become the input vector of the next layer and so on, until the last layer is reached. Because of this specific computation pattern, ANNs are also called *feed-forward neural networks*.

2.1.2 Stochastic Gradient Descent and Backpropagation

Considering the capability of a single perceptron as linear classifier, neural networks potentially can discriminate multi-class inputs, discriminating regions of any shape within the input space. The training process of such networks is based on the same idea of the perceptron one. Indeed, the key point of the algorithm is to *tune* the value of the weights in the network, according to the amount of the error committed with respect to the desired output.

This idea is implemented by the *Stochastic Gradient Descent* (SGD) algorithm. The intuition behind SGD is analogous to the perceptron update rule of the weights in Equation 2.3. Consider a *cost function* C defined as:

$$C(w) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, w) - t_n)^2 \tag{2.5}$$

where w represents the weights of the network, $y(x_n, w)$ is the output corresponding to $n\text{-}th$ sample in training set, and t_n is the associated desired output. Then, the goal is to minimize the error given by the cost function. In order to do so, the gradient $\nabla C(w)$ is computed with respect to each weight in the network. From an high-level perspective, this corresponds to compute the contribution that each weight gives to the final error. Recalling that the gradient is the direction of maximum growth of a function, the idea behind SGD is to iteratively change the value of the weights towards the opposite direction with respect to the gradient. The new value of the weights is computed according to the following rule:

$$w^{k+1} = w^k - \gamma \cdot \nabla C(w) \tag{2.6}$$

where k is the $k\text{-}th$ iteration and γ is the learning rate. It is worth to notice the similarity with the update rule of the single perceptron in Equation 2.3.

However, due to the presence of the hidden layers, it is necessary to find a way to compute the gradients of the weights for any layer in the network. Indeed, it is not possible to directly compute the derivative of the error with respect to the weights of the hidden neurons, since their output is not explicit in the expression of the cost function. This has been a major problem in research on NNs, until in 1986 an algorithm known as *backpropagation* was introduced in [31]. Nowadays, the backpropagation algorithm has become the standard approach for training of neural networks.

As explained in [32], starting from the last layer, it is possible to derive the expression of the derivatives of the cost function with respect to the weights of each layer in the network. Given the cost function $C(w)$ defined in Equation

2.5, the gradient can be written as:

$$\nabla C(w) = \frac{\partial C}{\partial a^L} \tag{2.7}$$

where L indicates the index of the last layer, and a^L is the output vector after the computation of the activation function. Then, exploiting the *chain rule*, it is possible to find the contribution to the error given by the weights in the last layer by computing the derivative of the activation funtion, providing:

$$\delta^L = \frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \cdot f'(z^L) \tag{2.8}$$

where w^L are the weights of the neurons in the output layer, while z^L is the output vector before the activation function is computed. This represents a measure of the error commited by the neurons in the last layer. With the same principle, it is possible to proceed *backwards*, and compute the derivative of the cost function with respect to the weights in any layer. The measure of the error for a generic layer l is computed as:

$$\delta^l = w^{l+1} \cdot \delta^{l+1} \cdot f'(z^l) \tag{2.9}$$

From this, it can be derived the equation to compute the derivative of any weight and bias in the network as:

$$\frac{\partial C}{\partial w^l} = a^{l-1} \cdot \delta^l \tag{2.10}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{2.11}$$

Backpropagation algorithm, together with SGD, provide an efficient method for training neural nets, which have become one of the most popular machine learning algorithm nowadays, outperforming existing solutions in many fields.

2.2 From Neural Networks to Computer Vision

Artificial Neural Networks demontrated their capability as classifiers even in image recognition tasks [11, 32, 33]. Indeed, it is quite straight-forward to consider an image as a vector of pixels, that become the input vector of a network. Although obtaining promising results, due to the complete interconnection between neurons, NNs are not able to take into account the instrinsic spatial locality of an image. However, taking inspiration from another biological process,

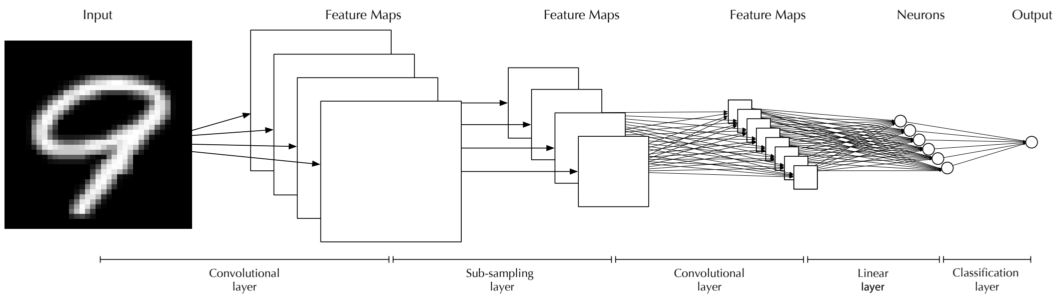

Figure 2.4: Common architecture of a CNN.

a variant of classic ANNs particularly efficient for image classification has been developed. This new type of neural networks are known as *Convolutional Neural Networks* (CNNs).

The work in [34], published in the late 1960s, showed that animal visual cortex contains specialized neurons that respond to small regions of the visual field. The same type of cells are present in similar regions across the visual cortex, providing a complete map of the visual space. This working principle lead to the idea behind CNNs structure.

As shown in [32], the input of the network can be considered as organized in a grid of input neurons, which particularly fits the 2D structure of an image. Then, each neuron in the next layer responds only to a small region of the input image, providing a *filter* made by the weights of the neuron. In this way, the output value can be considered as a pixel of the filtered image. The crucial point behind the mechanism of CNNs is that the weights are *shared* among the neurons in the layer, such that every output value is a pixel of the same filtered image that is known as *feature map*.

The sharing of the weights exploits the spatial locality of an image, as it happens in the animal visual cortex. In fact, the purpose of the layers in a CNN is to find correlation in different regions of the image, extracting *features* from it. In the same way of ANNs, the weights of the neurons are the result of the training process, providing *filters* that are able to recognize distinct characteristics, such as edges, lines, circles and so on (Figure 2.5, from [1]). Moving forward in the network topology, the filters are able to extract more complex and abstract features.

Those features are then aggregated and utilized in order to provide a clas-

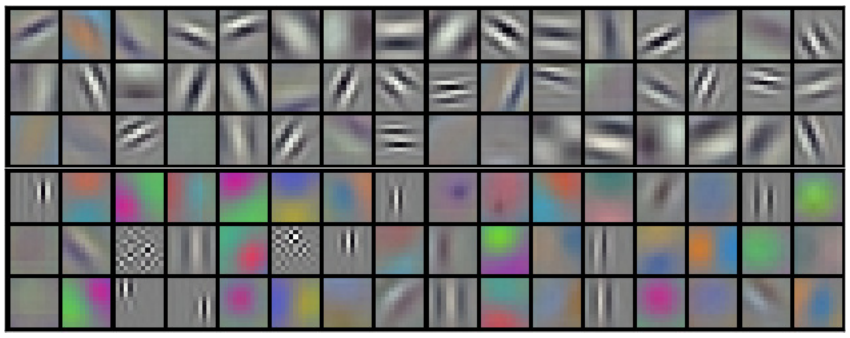

Figure 2.5: Features from the first layer of AlexNet. Image from Krizhevsky et al. [1].

sification of the input image. Figure 2.4 presents the overall architecture of a CNN; it is structured in a configurable chain of layers that can be partitioned in two main stages, called *feature extractor* and *classifier*, respectively.

2.3 Topology of Convolutional Neural Networks

The feature extractor is the defining part of a CNN. In general, it is composed of a sequence of the so-called *convolutional layers*, usually followed by *activation layers* and alterned with *pooling* or *sub-sampling layers*. The classifier instead is constituted by *fully-connected layers*, which form a standard ANN in the last part of the network.

CNNs demonstrated to be very efficient in image classification. Nowadays, they have become the state-of-the-art in image recognition and classification, and they are widely used in many applications in the field of Computer Vision. The next subsections provide the details on the structure and the implementation of the different types of layers of a CNN.

2.3.1 Convolutional layer

The *convolutional layer* implements an arbitrary number of K filters, or *kernels*, to extract relevant features from the input image; for each filter k, a correspoding output feature map is generated. Usually, the dimensions of the kernels are small, varying from 3x3 to 5x5 pixels or 12x12 in case of very big input images.

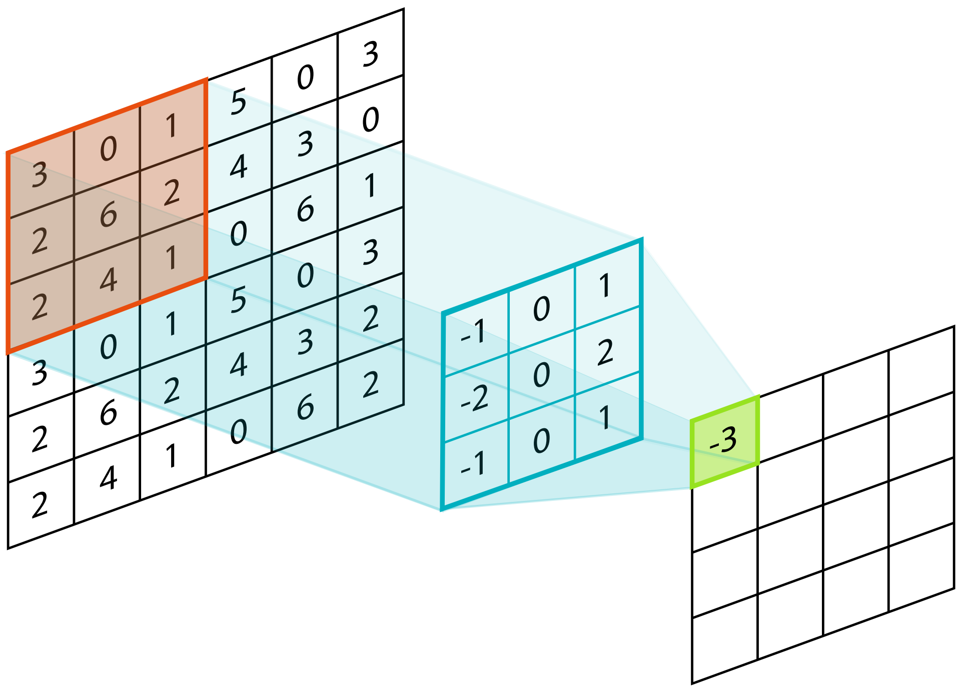

Figure 2.6: Computation pattern of 2D convolution.

The mathematical operation implemetented by each kernel is a convolution of the volume made by the input pixels with the kernel weights. The input volume has shape $Ch \times H \times W$, respectively the *channels* (e.g. RGB), the *height* and the *width* of the image. The following equation shows how each output pixel is computed.

$$o^k_{i,j} = \sum_{c=0}^{Ch} \sum_{h=0}^{K_H} \sum_{w=0}^{K_W} (w^k_{h,w,c} \cdot x_{i+h,j+w,c}) + b_k \tag{2.12}$$

where i and j are the coordinates of the output pixel, k is the k-th filter with dimensions $Ch \times K_H \times K_W$, x and w are respectively the input pixel and the weight, and b_k is the bias of the kernel.

From an abstract point of view, filters are slided on the image to produce the output feature map. The sliding factor, called *stride*, is usually set to 1, but sometimes it has higher values in order to significantly reduce the input dimensions, especially in the earlier stages of the network. In fact, every position of the filter computes one single pixel in output, as depicted in Figure 2.6. The dimensions of the output feature map are given by:

$$\begin{aligned} H' &= \frac{H - K_H}{K_{stride}} + 1 \\ W' &= \frac{W - K_W}{K_{stride}} + 1 \end{aligned} \tag{2.13}$$

A convolutional layer can be implemented in an algorithmic way given by the following pseudo-code:

```
for (i = 0; i < H - Hk + 1; i++)
  for (j = 0; j < W - Wk + 1; j++)
    for (k = 0; k < K; k++)
      for (c = 0; c < Ch; c++)
        for (h = 0; h < KH; h++)
          for (w = 0; t < KW; w++) {
            w = weights[k][c][h][w];
            x = img[i+h][j+w][c];
            o[k][i][j] += w * x;
          }
```

Listing 2.1: Pseudo-code of a convolutional layer

It is worth noting the lack of control structure in the code. This makes the specific computation pattern of the convolutional layer to be extremely *dataflow*.

2.3.2 Pooling layer

The *pooling* or *sub-sampling* layer is generally inserted between two convolutional ones to progressively decrease the size of the elaborated data. The working principle of this layer is similar to the convolutional layer one. Indeed, the pooling layer is implemented as a small filter (usually 2x2 or 4x4 for large images) that selects or compute one output pixel according to an operator. The most common type of operator is the so-called *Max-Pooling*, which selects the pixel in the filter with maximum value. Other types of sub-sampling operators are the *Min-Pooling* and the *Mean-Pooling*, which select the minimum value and compute the mean of the input pixels, respectively. After the pooling layer, the dimensions of the output feature maps are reduced according to Equation

2.13. However, in general the pooling filter dimension is equal to the stride, also called *pooling step*, providing the following equations for the new feature maps size:

$$\begin{aligned} H' &= \frac{H}{P_{step}} \\ W' &= \frac{W}{P_{step}} \end{aligned} \tag{2.14}$$

There are two main purposes of this type of layer. On the one hand, reducing the dimensions of the feature maps allows to save significant memory to store the intermediate results. On the other hand, pixel selection makes sure that only the most relevant features will be forwarded to the next layers.

2.3.3 Activation layer

The activation layer is an element-wise operator that applies an activation function to each of the input pixels, in the same way of what happens in the perceptron. The application of the activation function allows to *squash* the value of the pixels within the boundaries of the specific function, avoiding indefinitly increasing of the values due to the *multiply-accumulate* operation in the convolution layer. Moreover, such function add a *non-linearity* that smooths the obtained classification boundaries. Common functions used in the activation layer are the following:

- Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$ with range $[0, 1]$
- Hyberbolic tangent function: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ with range $[-1, 1]$

2.3.4 Fully-connected layer

The fully-connected layer, also called *linear layer*, is the same type of layer of a classical ANN. Basically, it is composed of several perceptron units that take the output of the previous layer as input. It is worth noting that the activation function of the perceptron in case of fully-connected layers is usually the identity function. In fact, the same result can be obtained by putting an activation layer afterwards.

From a mathematical point of view, linear layers compute the sum of the product of the weights and the inputs, given by the following equation:

$$\mathbf{y} = W \cdot \mathbf{x} \tag{2.15}$$

where **y** is the vector obtained by the dot-product of the matrix of the weights W, and the input vector **x**. Each element $w_{j,i}$ in W is the weight between the i-th input neuron and the j-th neuron in the layer.

As it happens for the convolutional layers, even fully-connected layers have a specific computation pattern that makes them highly-suitable for hardware acceleration, thanks to the absence of control structure. The pseudo-code of a fully connected layer is the following:

```
for (j = 0; j < OUT_NEURONS; j++)
  for (i = 0; i < INPUT_NEURONS; i++) {
    w = weights[j][i];
    y[j] += w * x[i];
  }
```

Listing 2.2: Pseudo-code of a fully-connected layer

The number of neurons of the last linear layer is equal to the number of classes to be recognized. In this way, it is possible to apply a normalization operator that gives a probabilistic interpretation of the provided classification. In particular, one of the most common operators is the *SoftMax* operator σ:

$$\sigma_i = \frac{e^{y_i}}{\sum_{n=1}^{N} e^{y_n}} \qquad \forall j \in [1, N] \tag{2.16}$$

where y_i is the output vector generated by the linear layer. This operator enforces the N values of the output to lie in range $[0, 1]$ and to sum up to 1, such that they can be interpreted as the probability of the input to belong to a certain class.

2.4 Convolutional Neural Networks in Research

Since their first proposal in the late nineties in [11], CNNs have been employed in many fields due to their capability of identifying different and complex features in images. In [11], LeCun et al. first proposed a CNN-based approach for the recognition of handwritten digits of the MNIST dataset [28], composed of 28x28 black and white images (Figure 2.7). Thanks to its popularity, this dataset has become one of the most adopted benchmark for CNNs related works. The models proposed in the literature have been able to obtain a classification accuracy up to the 99,7% of this testing dataset [35, 36], composed of 10 000 images. This is an incredible result, especially considering that some of the numbers are very difficult to be recognized even from humans.

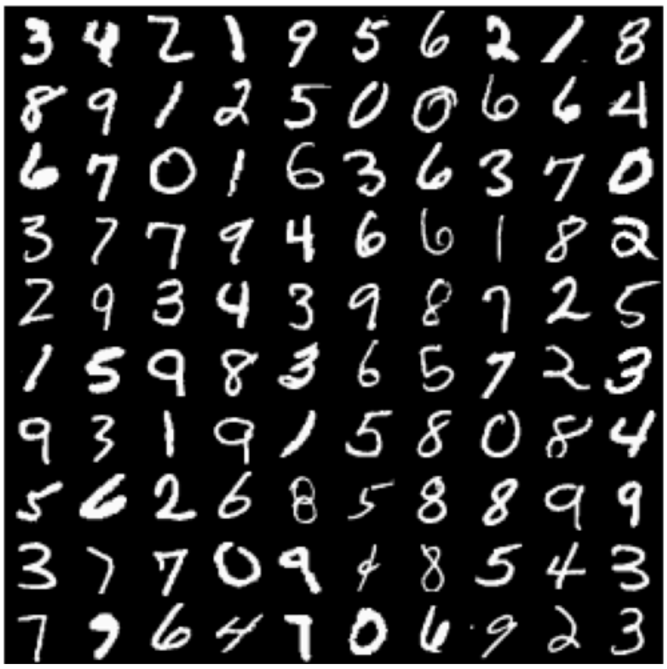

Figure 2.7: Sample of the handwritten digits belonging to the MNIST dataset. Image from: `http://theanets.readthedocs.io`.

Interest in this kind of algorithms has been growning constantly, and other more complex datasets have been introduced as benchmarks, such as the Cifar-10 and the Cifar-100 [29], composed of 32x32 RGB images belonging to 10 and 100 different classes, respectively (Figure 2.8). Another well-known dataset is the ImageNet dataset [37], which is the object of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [38]. This dataset is composed of big images divided in high-level catergories, sub-categories and so on. The ILSVRC is one of the largest competitions in Computer Vision and every year teams compete to claim the state-of-the-art performance on the dataset. CNNs demonstrated to be very effective in the classification task of those images. In fact, the popular network known as AlexNet, presented in [1], was able to obtain 62.5% and 83% in top-1 and top-5 accuracy, winning the ILSVRC competition in 2012.

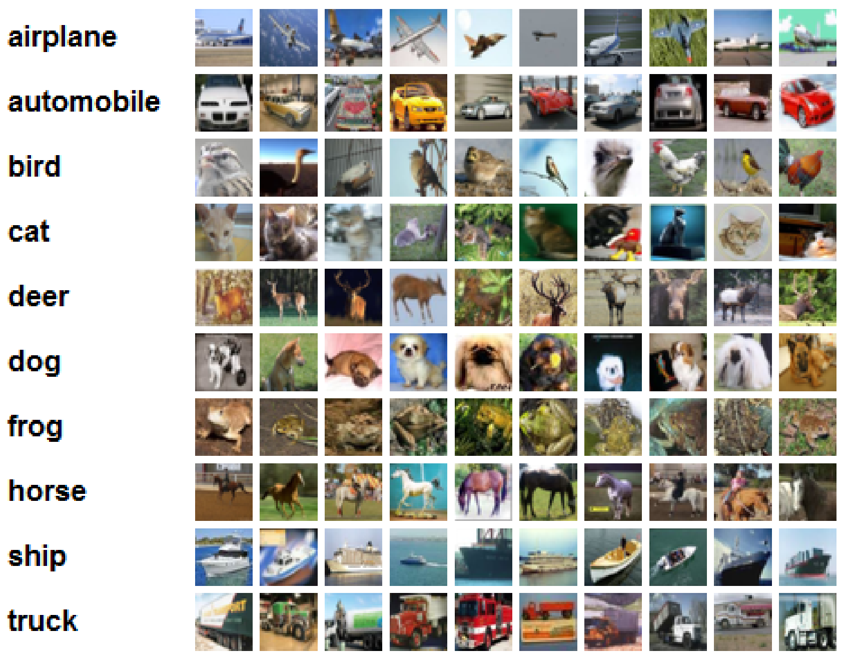

FIGURE 2.8: Sample RGB images belonging to the Cifar-10 dataset. Image from: `https://www.cs.toronto.edu/kriz/cifar.html`

Since then, new CNN models have been proposed to further improve the classification accuracy, involving also companies as Google and Microsoft. In particular, in 2014 Google won the ILSVRC challenge presenting a network name GoogLeNet [17], which is able to achieve 93.33% top-5 accuracy on the test set. Then, in 2015 Microsoft proposed its own model called ResNet [39], which got the first place in the same competion, obtaining a top-5 accuracy of 96.43%.

Thanks to their capability in image classification, CNNs started to be employed in many computer vision applications. As illustrated in [40], such networks have been firstly adopted in *Optical Character Recognition* (OCR) and hanwriting recognition systems, such as the works in [41, 42, 43], including also case studies for Chines and Arabic characters [44, 45].

CNNs have also been used for object detection in images, even in case of face recognition, providing very high accuracy and real-time performance [46, 47, 48, 49]. Examples of such applications include the work developed by Google in [50] for detecting faces and license plate in StreetView images for privacy purposes. NEC deployed a CNN-based sysem able to track customers in supermarket and recognizing gender and age. Google introduced image classification in the search engine services, through the Search by Image feature. Amazon developed deep-learning based technologies for image recognition, face detection and sentiment analysis in Amazon Web Services (AWS), and provides APIs for such applications with AWS Rekognition [2].

Other experimental applications using CNNs also include hands/gesture detection [51, 52], logos and text recognition [53], but also vision-based obstacle avoidance for off-road mobile robotos [54]. Moreover, CNNs are emerging also for another type of multimedia data, with examples of natural language processing applications [55, 56, 57, 58].

2.5 Hardware Acceleration of CNNs

One of the main challenges application engineers have to face with CNNs is their considerable computational demand as the size of the network grows to achieve the required accuracy. As a matter of fact, most of modern CNN applications achieve bad performance on *Central Processing Unit* (CPU), even in the case when multi-core architectures are used. For this reason, a large research effort has been devoted in the past years to the acceleration of CNN by means of dedicated co-processing units such as GPUs, FPGAs or even custom ASIC modules. GPUs have demonstrated to achieve adequate performance levels (e.g. [1, 59, 60]); however, their considerable power consumption is not affordable in application scenarios such as the embedded and mobile one. On the other hand, even if offering the best performance/power consumption tradeoff, ASIC solutions (such as [61]) are not very appealing due to their high design and manufacturing costs. In the end, the literature has shown in many works (e.g. [62, 63, 20, 64, 65, 66]) how FPGAs represent the most suitable device for acceleration thanks to their highly-parallel and power-efficient grid of programmable processing elements.

In [67], the author developed a DRAM simulator based on memory access patterns of CNNs. A memory-centric design method for CNN accelerators on

FPGAs is presented in [18]. The efficient data pattern access proposed by the authors resulted in a minimization of on-chip memory size and improvement in data reuse, as well as better performance and energy efficiency. This work was evaluated on a Xilinx Virtex-6 FPGA board and it outperformed standard scratchpad memories accelerators.

The work reported in [63] aimed at increasing performance and reducing energy consumption of CNN accelerators on FPGA thanks to computation reordering and local buffer usage. Moreover, the authors proposed a novel analytical methodology whose purpose was to optimize nested loops for inter-tile data reuse. This work evaluation proved a significant boost in Xilinx MicroBlaze soft-core performance, as well as a 2.1X reduction in data movement.

The work reported in [20] exploits the *roofline model* [68] in order to perform a design space exploration for the acceleration of CNN convolutional layers on FPGA. The defined architecture is based on a single module executing in sequence all the layers. The resulting implementation of the AlexNet CNN [1] outperformed the previous state-of-the-art implementation.

A programmable CNN processor implemented on a custom board powered by a low-end DSP-oriented FPGA, called *ConvNets Processor* (CNP), is presented in [69]. Such processor relies on an external memory module and takes in input a sequence of instructions generated by a *Lush*-based network compiler, which accepts a Lush description of the CNN.

In [65], the authors propose a CNN accelerator design to overcome resource underutilization on FPGA devices. The result is a modular implementation of the CNN consisting of a chain of various modules supporting a pipelined execution; the aim is to optimize the resources and avoid data dependencies among them. The authors achieved on a Virtex-7 485T FPGA a 1.3X higher throughput than the single module state-of-the-art design of the same network, with a 97.1% of dynamic resource utilization.

The authors of [66] present a CNN accelerator design for Image-Net large-scale image classification on FPGA devices. The authors analyzed the literature on CNN designs to show that convolutional layers are computational-centric, whereas linear layers are memory-centric. Then, they developed a dynamic-precision data quantization method to enhance bandwidth and resource utilization. The experimental results reported only 0.4% accuracy loss, while the average performance remarkably overcomes the previous approaches.

All the approaches discussed so far focused on the architectural design of the CNN accelerator in order to improve performance and power consumption. However, authors performed the design activities manually and, most of the time, the implemented solutions are specifically targeted for a single application scenario, such as the HPC one. This implies that the re-targeting and adaptation of such HW design to different scenarios, such as the embedded one, is a demanding and time consuming activity. The goal of this work is to implement a novel framework overcoming these limitations by proposing new re-usable architectural templates that are automatically generated, targeting different scenarios.

The Proposed Methodology 3

This Chapter describes the methodology proposed in this thesis. Section 3.1 gives an overview of the framework, while the next sections provide the details on the different modules. Section 3.2 analyse the Intermediate Representation used in the framework, as well as the conversion from an existing Caffe description of a CNN. Then, Section 3.3 explains the training process, illustrating the supported dataset formats and describing the training strategy. Finally, Section 3.4 provides the details on the generation of the C++ code for the High Level Synthesis.

3.1 Framework Overview

As stated in Chapter 1, CNNs are a very widely discussed topic in research and engineering. Lot of efforts have been done in order to provide improvement in accuracy, performance and energy efficiency of such algorithms. Despite the latest achievements in this field, the huge amount of operations and memory required for the parameters of a CNN makes it a very difficult task. This is particularly true when this type of networks have to be implemented in hardware and/or scenarios critical as regards power consumption, such as embedded devices and datacenters.

Even though GPUs demonstrated to be efficient as accelerators for both training and inference of CNNs, such devices are very power hungry, which makes them not suitable in the aforementioned power-constrained scenarios. For this reason, interest in FPGAs as accelerators has been growing in the last years. Indeed, this kind of devices are extremely low-power, and their architecture perfectly fit the massively parallel computation pattern of CNNs and neural networks in general.

However, the process to design and implement such algorithms on FPGAs can be complex and time consuming, especially for developers who are not versed in hardware design. Despite many machine learning frameworks have been proposed targeting both CPUs and GPUs as underlying device (e.g. TensorFlow [12], Caffe [13]), there are still no tools that allow to reduce the development effort to implement CNNs on FPGA-based accelerators.

Given these motivations, the aim of this work is to provide a framework that significantly reduce the development time, providing a toolchain for fast-prototyping and deployment of hardware accelerators for inference.

The modules of the framework and the exposed APIs have been written in Python. There are two main reasons behind this specific implementation choice. On the one hand, almost all machine learning frameworks provide Python APIs, giving the possibility to easily integrate functionalities where needed. On the other hand, the framework is specifically designed as a set of Python modules, exposing the APIs that provide the toolchain for the automatic generation of the C++ code of the CNN. The modularity of the framework makes also the framework to be scalable and provides a flexible external interface. In fact, it would

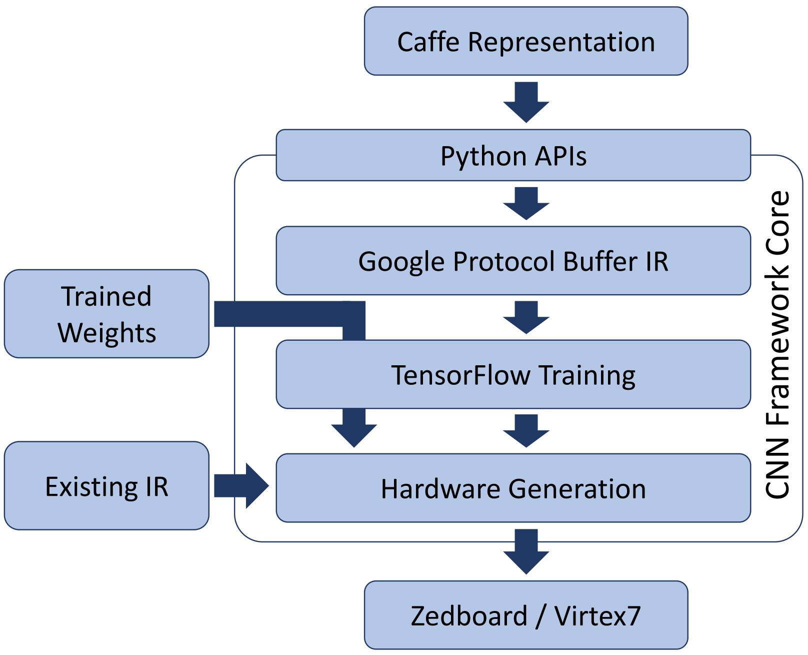

FIGURE 3.1: Framework organization and workflow.

be straight-forward to both use the provided libraries as a Python package, for instance using Jupyter Notebooks for enhanced code visualization, and build a GUI on top of it. Moreover, each of the modules can be used as a stand-alone Python script.

The overall organization of the framework is provided in Figure 3.1. As it can be seen, there are three main modules provided:

- PROTOBUF GENERATOR: this module provides the APIs for both the conversions from the Caffe `.prototxt` definition of a network, or a custom JSON notation. The output generated by this module is a `.proto` file, containing the serialized object definition made with the Google Protocol Buffer [70].

- TRAINER: this module is responsible of the training of the specified CNN using TensorFlow APIs [12]. The datasets for training and testing are required as input, as well as the `.proto` definition of the specified model, either generated by the PROTOBUF GENERATOR module, or created from scratch with Protocol Buffer APIs. The TRAINER will generate the files containing the trained weights of the CNN as output.

- Hardware Generator: this module provides functions to generate both the C++ source code and the `.tcl` scripts for High Level Synthesis and bitstream generation using Vivado and Vivado HLS [27]. The same `.proto` definition is used as model to generate the code and to set the target device for synthesis.

The next sections provide the details on the framework modules and their implementation. In particular, a description of the utilized Protocol Buffer definition and the training stratergy are provided.

3.2 The Intermediate Representation

The *Intermediate Representation* (IR) of the model is a key point within the framework infrastructure, since every module relies on this specification to perform its functionalities. In order to provide the required flexibility to store all the types of information needed by the different modules, it has been decided to use Google Protocol Buffers [70], or simply *ProtoBufs*, as IR.

ProtoBufs provide a light-weight, extensible mechanism to store and serialize data. In order to be used, ProtoBufs require a file containing the specification of the information to be represented. The information inside a ProtoBuf is structured in a so-called *message*, in which data are represented as fields in the message definition. The fields can be either standard data-types as `int`, `bool` and `string`, enumerations, or another `message` type. Once defined, the specification of ProtoBufs has to be compiled, providing a special generated code with APIs to write, read and manipulate messages as objects.

Furthermore, the choice of using Protocol Buffers as IR allows to provide the same structure of models generated by the Caffe framework, which also relies on this mechanism. Indeed, there are different approaches that can be used as *entry-points* to the framework, namely, either a Caffe `.prototxt` model or a JSON file can be provided to the ProtoBuf Generator, as shown in Figure 3.1.

3.2.1 Protocol Buffer definition

The top-level `message` definition within the framework is the one provided in Listing 3.1. As it can be seen from the code, the `Project` message provides a container for all the informations needed by different modules. In particular, the ProtoBuf contains information about the network model, the training and testing datasets, the parameters for training, and the target FPGA device for the HLS and the hardware synthesis.

```
message Project{
    optional string name = 1;
    optional NetParameter network = 2;
    optional Dataset train_set = 5;
    optional Dataset test_set = 6;
    optional Training train_param = 7;

    enum DeviceType{
        ZEDBOARD = 0;
        ZYBO = 1;
        VIRTEX7 = 2;
    }

    optional DeviceType device = 3 [default = ZEDBOARD];
    optional uint32 num_cores = 4 [default = 1];
}
```

Listing 3.1: Protocol Buffer definition of a `Project` message.

The network model definition is described in Listing 3.2. The `NetParameter` message is compliant with the Caffe definition of the network. Indeed, the same definitions for the different type of layers are used. Although reduced, the framework definition of the network model supports the primary types of layer of a CNN, i.e. convolutional layers, pooling layers and fully-connected layers. It is also worth noting that the topology of the network is given by a sequence of generic `LayerParameter` messages, providing a scalable definition in case of additional supported types of layer.

```
message NetParameter {
    optional string name = 1; // representative name of the network
    // The layers that make up the net.  Each of their configurations, including
    // connectivity and behavior, is specified as a LayerParameter.
    repeated LayerParameter layer = 2;
}

message LayerParameter {
    optional string name = 1; // the layer name
    optional string type = 2; // the layer type
    optional string bottom = 3; // the name of each bottom blob
    optional string top = 4; // the name of each top blob

    // Layer type-specific parameters.
    optional ConvolutionParameter convolution_param = 106;
    optional PoolingParameter pooling_param = 121;
    optional InnerProductParameter inner_product_param = 117;
    optional MemoryDataParameter memory_data_param = 119;
}
```

Listing 3.2: Protocol Buffer definition the network model, compliant with a subset of the one provided by the Caffe deep learning framework.

The message definitions of the supported types of layers are provided in Listing 3.3. The hyper-parameters of each type of layer are specified as fields. In particular, the `MemoryDataParameter` is one of the available Caffe definitions for the input data layer, containing information on the input image dimensions. The `ConvolutionParameter` provides information on the filters of a convolutional layer, i.e. the number of output feature maps and the kernels size. Then, the `PoolingParameter` defines the structure of the pooling layer, including the subsampling operator for future support to different types of pooling. Finally, the `InnerProductParameter` specifies the structure of a fully-connected layers by providing the number of output neurons of the layer. It is worth to notice that some of the parameters may seem redundant, however, this guarantees compatibility with older definitions of existing network.

```
message ConvolutionParameter {
    optional uint32 num_output = 1; // The number of outputs for the layer
    optional bool bias_term = 2 [default = true]; // whether to have bias terms

    repeated uint32 kernel_size = 4; // The kernel size
    repeated uint32 stride = 6; // The stride; defaults to 1
    optional uint32 kernel_h = 11; // The kernel height
    optional uint32 kernel_w = 12; // The kernel width
}

message PoolingParameter {
    enum PoolMethod {
        MAX = 0;
        AVE = 1;
    }

    optional PoolMethod pool = 1 [default = MAX]; // The pooling method
    optional uint32 kernel_size = 2; // The kernel size (square)
    optional uint32 stride = 3 [default = 2]; // The stride (equal in Y, X)
    optional uint32 kernel_h = 5; // The kernel height
    optional uint32 kernel_w = 6; // The kernel width
}

message InnerProductParameter {
    optional uint32 num_output = 1; // The number of outputs for the layer
    optional bool bias_term = 2 [default = true]; // whether to have bias terms
}

message MemoryDataParameter {
    optional uint32 batch_size = 1;
    optional uint32 channels = 2;
    optional uint32 height = 3;
    optional uint32 width = 4;
}
```

Listing 3.3: Protocol Buffer definition of `Layer` messages.

As regards the `Dataset` message definition, it contains the information about the images, as well as the format of the files. The currently supported formats are the `IDX` format used for the MNIST dataset [28], and the most common types of image format as `png`, `jpg`, `bmp` and so on. The message definition is provided in Listing 3.4.

```
message Dataset {
    optional string path = 1;
    optional string img_file = 8;
    optional string img_ext = 10;
    optional string label_file = 9;

    enum Format{
        MNIST = 0;
        OTHER = 1;
    }

    optional Format format = 2 [default = OTHER];
    optional ImageInfo img_info = 6;
    optional uint32 num_images = 3;
    optional uint32 classes = 7;
}

message ImageInfo {
    optional uint32 channels = 4;
    optional uint32 height = 5;
    optional uint32 width = 6;
}
```

Listing 3.4: Protocol Buffer definition of a `Dataset` message.

Finally, each specific `Project` message can be serialized or parsed as an object to be used by the framework modules. It is also possible to export the message in a human-readable string represention in a `.prototxt` file.

3.3 The Training process

The Trainer is the module providing APIs to perform the training of the specified CNN. In addition to training, this module is able to export the weights in a convenient file formats, namely the `csv` or the `npy` file format from the `numpy` Python library.

The purpose of this module is to provide a first implementation of the CNN training process in order to obtain the weights needed for inference. Indeed, it is not possible to determine an optimal training strategy for all the possible network topologies in the design space of CNN. However, it is possible to implement a training approach that is able to obtain a sufficiently good accuracy for the given model.

The training flow provided by the Trainer module is composed as follows. Firstly, the datasets for the training and testing are loaded into the `Trainer`, as well as the corresponding labels. As stated in Section 3.2, the supported formats of the images can be either the IDX format as for the MNIST dataset, or one of the standard image formats (e.g. `png`, `bmp`, etc.).

Secondly, the module builds the model of the CNN according to the structure parsed by the serialized ProtoBuf network message. Thirdly, the model is mapped to a computational graph definition within a TensorFlow session. A `Graph` in TensorFlow is a set of nodes representing operations among tensors. For each convolutional or fully-connected layer in the network topology, the weights and the bias are instantiated as TensorFlow `Variable`, which are tensors that can be trained by the backpropagation algorithm. Then, for each type of layer an operation is added to the computational graph.

At this point, the TensorFlow session is run to compute both the feedforward and the backward passes, updating the weights of the network. The batch size of images and the learning rate are set according to the ProtoBuf message of the training parameters.

The loss function used to compute the gradients is the *cross-entropy* error function, applied after a *SoftMax* operator as described in Chapter 2. The cross-entropy loss function is defined by the following equation:

$$C(w) = -\frac{1}{N}\sum_n \sum_j (t_j \ln(a_j^L) + (1 - t_j)\ln(1 - a_j^L)) \tag{3.1}$$

where N is the batch size, j is the index of the $j\text{-}th$ output neuron, t_j is the correct output for that neuron, while a_j^L is the actual output. As described in [32], the cross-entropy function has nice properties as a cost function. In particular, it allows to prevent the learning slowdown caused by the derivative of the activation function $\sigma'(z)$. Indeed, thanks to its mathematical definition provided in Equation 3.1, the derivative of the cross-entropy with respect to the weights $\frac{\partial C}{\partial w}$ is proportional to the error committed in the output, i.e. the larger the error, the faster the neuron will learn, according to the update rule provided in 2.6.

Finally, after the training process is complete, the TensorFlow variables representing the weights are evaluated as `numpy` arrays and exported in the specified format for the next step of the framework work-flow.

3.4 Hardware generation

The Hardware Generator module is responsible for the main purpose of the framework: to automatically generate the source code of a CNN suitable for High Level Synthesis and implementation on an FPGA accelerator.

The functions provided by the module perform the following three tasks:

- A `write_hls` function is provided in order to generate the C++ code implementing the CNN topology. This is done by parsing the ProtoBuf description of the model as it happens for the training process. For each layer in the network, an hardware module is instantiated by using templatized functions with the dimensions of the layer as parameters. Furthermore, also the `pragmas` to configure the hardware generated by the HLS process are added to the code at this stage. The details on the hardware implementation are provided in Chapter 4.

- Along with the C++ implementation of the network, the module provide functions to read the weights generated by the training process, or any other weights file compliant with the format given by the `Trainer` module. The imported weights are then written to an header file as static multi-dimensional arrays and initialized.

- Lastly, the module provides APIs to generate the `tcl` scripts to perform the High Level Synthesis and the hardware implementation down to the generation of the bitstream to configure the FPGA. The scripts are in charge to create the projects for Vivado HLS and Vivado, add the source code, generate the block design and run all the steps to write the bist-stream. This allows to save a lot of effort and time to set up the tools every time a new implementation is needed.

The Proposed Hardware Template 4

This chapter gives the details on the architecture of the hardware accelerator that implements a CNN. Section 4.1 explains the overall structure of the CNN accelerator, the interconnections between the layer modules and the interfaces with the sorrounding system. The following sections provide the implementations of the different type of layers as stand-alone modules. In particular, Section 4.2 describes the architecture of the convolutional layer module, Section 4.3 details the max-pooling layer, while Section 4.4 provides the implementation of the fully-connected layer. Finally, Section 4.5 concludes the chapter explaining the overall architecture of the hardware accelerator implemented on the FPGA, including the components for memory communication and run-time control.

4.1 CNN Accelerator

As stated in the introductory chapter, the specific computation patterns of CNNs makes them highly suitable for hardware acceleration. The main reason under this statement is that the core computation of both the convolutional and the fully-connected layers, which are the most computationally-intensive, can be reduced to a *dot-product* between the matrix of the weights W and the input vector $\mathbf{x}$. For the fully-connected layer, this is trivial to be seen from Equation 2.15. As regards the convolutional layer instead, from Equation 2.12 we can consider the matrix W_c^k as the weighted filter related to the c-th input feature map and the k-th output feature map, thus it can be seen as subsequent dot-product operations to reconstruct the 3D convolution. This kind of computation can be efficiently mapped in specialized hardware like FPGAs and GPUs, which exploits the instrinsic parallelism of the dot-product to obtain a speed up in terms of latency of a single multiply-accumulate operation.

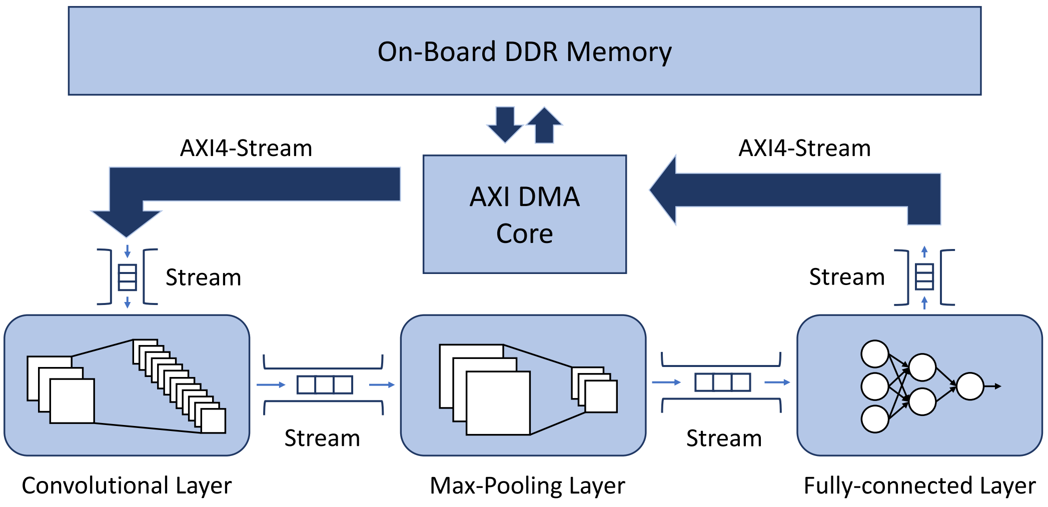

Figure 4.1: Convolutional Neural Network architetcural template.

The hardware accelerator that the framework is able to generate from the network description is depicted in Figure 4.1. Each layer in the CNN topology is implemented as a single module with connections implemented as *First-In First-Outs* (FIFOs) for input and output. This choice is specifically tailored to build a *data-flow* architecture that allows to process images in a streaming way.

The modules of the accelerator form a *pipeline* in which each layer is a different stage. The implementations of the different layers guarantee that each pixel has the lowest possible latency inside each module, so that the throughput of the overall accelerator is maximized.

The code shown in Listing 4.1 represents a simplified version of the CNN accelerator of an example network made by one convolutional layer with max-pooling, and one fully-connected layer. As it can be seen, each layer is implemented as an instance of a templatized C++ function, in which the dimensions of the layer are provided as macros that are automatically generated by the Hardware Generator module described in Section 3.4. The `pragmas` in the code determine the directives to guide the HLS process and obtain the desired hardware implementation of the accelerator.

```
void cnn(hls::stream<ap_uint<DATAWIDTH> > &axistream_in,
         hls::stream<ap_uint<DATAWIDTH> > &axistream_out){
#pragma HLS INTERFACE ap_ctrl_none
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITION variable=W_conv1 complete
#pragma HLS ARRAY_PARTITION variable=W_fc1 dim=2

    hls::stream<data_t> data_stream;
    hls::stream<data_t> conv1_stream;
    hls::stream<data_t> pool1_stream;
    hls::stream<data_t> fc1_stream;

    convLayer<CHANNELS, FMOUT_1, KERSIZE_1, IMG_DIM, data_t>
        (data_stream, conv1_stream, W_conv1, b_conv1);
    maxPoolLayer<FMOUT_1, POOLSIZE_1, FMDIM_1, FMDIM_1, data_t>
        (conv1_stream, pool1_stream);
    fullyConnLayer<FC_NEURONS_0, FC_NEURONS_1, data_t>
        (pool1_stream, fc1_stream, W_fc1, b_fc1);
}
```

Listing 4.1: C++ implementation of the CNN accelerator top function.

There are some aspects that are worth noting from the code. Firstly, the interface of the function takes two `axistreams` for input and output. This particular modules implement FIFOs according to the *AXI4-Stream* protocol, which provides a very low overhead for control signals, allowing streaming data transfer. Furthermore, the external interface is set to `ap_ctrl_none`, which removes the control signals of the accelerator. In this way, the accelerator is always *active* and it starts to compute as soon as the input FIFO begins to fill, allowing also to send batch of images without having to read the results of each classification and to restart the accelerator.

Secondly, the `DATAFLOW` directive makes Vivado HLS build the pipeline composed of the layer modules. So, as it happens for the external interface, each subsequent module can start to process data as soon as it gets the first pixel given by the previous layer in output, avoiding to stall until the computation of the previous module is completed and thus maximizing the throughput.

Thirdly, due to the `ARRAY_PARTITION` directives, the arrays that store the weights for both the convolutional and the fully-connected layers are partitioned on the dimension specified in the `pragma`. Partioning an array means to split the array in the elements of the specified axis and to store them in separated memory locations, either implemented in *Look-Up Tables* (LUTs) or *Block-RAMs* (BRAMs) in the programmable logic of the FPGA.

Finally, we can see the declaration of the FIFOs for input and output of each layer. FIFOs are implemented as `hls::stream`, a template class provided by the Vivado HLS library. The template parameter specifies the data type contained in the FIFO, while the class provides methods for read and write data and allows to set the *depth* of the FIFO to synchronize the data exchange between two modules.

4.2 Convolutional Layer Accelerator

The convolutional layer module implements the pseudo-code reported in Listing 2.1. The hardware implementation of the convolutional layer exploits the instrinsic parallelism in its computation pattern. Indeed, the filters can operate independently of each other, providing one pixel per output feature map at each clock cycle.

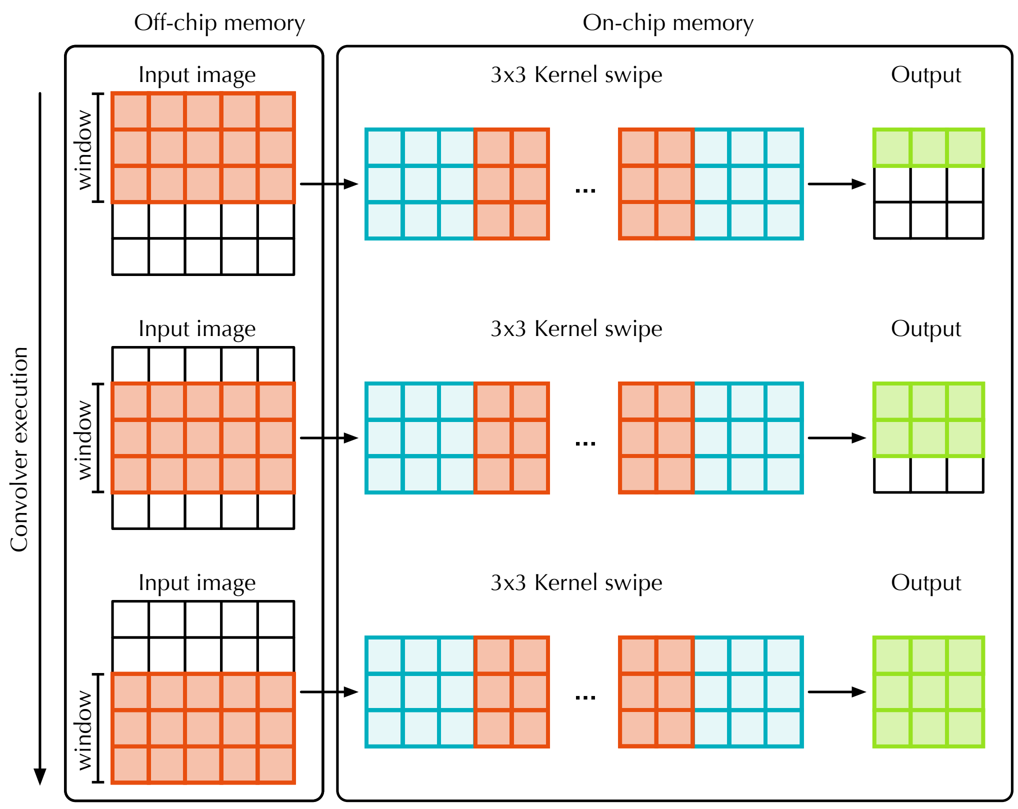

FIGURE 4.2: Example of input image loading on on-chip memory

However, the overlap between different positions of the same filter on the input feature map makes necessary to store at least some of the input pixels among different iterations. For this reason, at the beginning of each convolutional layer a shift register is present to store only a portion of the data needed for the computation of a row of the output feature map. The shift register has been realized using the `hls::Window` component from the Vivado HLS library.

This component allows to store data in a bidimensional array which is completely partioned. The 2D window allows to insert a new value anywhere in the matrix with a direct access to the memory location of the pixel. The window permits to shift the values in the matrix in two different ways: on the one hand, it is possible to shift the values on a row (or a column) left or right, on the other hand, all the rows (or the columns) can be shifted together up or down. Moreover, the partioning of the window allows to access all the pixels in the matrix in parallel.

Such component is configured to store a number of rows of the input equals to the height of the convolutional kernels K_H, and a number of columns equal to $W \times Ch$, where W and Ch are respectively the width and the number of the input feature maps. Figure 4.2 illustrates how the window behaves in a convolutional layer. At the beginning, the first K_H rows of the window are filled in order to have enough data to start the computation. After the kernel swipes over the entire row, the content of the window shifts up, deleting the pixels of the first row that are no more needed. At this point a new row is loaded into the window for the next iteration, and the computation continues in this way until all the output feature maps are produced.

```
void convLayer (hls::stream<data_t> &streamIn, hls::stream<data_t> &streamOut,
    data_t weights[FM_OUT][CH][KH][KW], data_t bias[FM_OUT]){
    for (int i = 0; i < DIMH-KH+1; i++) {
        for (int j = 0; j < DIMW-KW+1; j++) {
            for (int k = 0; k < FM_OUT; k++) {
#pragma HLS PIPELINE
                for (int ch = 0; ch < CH; ch++) {
#pragma HLS UNROLL
                    for (int kh = 0; kh < KH; kh++) {
#pragma HLS UNROLL
                        for (int kw = 0; kw < KW; kw++) {
#pragma HLS UNROLL
                            data_t w = weights[k][ch][kh][kw];
                            data_t pixel = window.next();
                            acc += w * pixel;
                        }
                    }
                }
                streamOut.write(acc + bias[k]);
            }
        }
        window.shiftUp();
    }
}
```

Listing 4.2: C++ implementation of the convolutional layer module.

Listing 4.2 reports the C++ code for the convolutional layer. As it can be seen from the code, the computation pattern is very similar to the pseudo-code reported in chapter 2. However, the presence of the `pragmas` determines how the module will be implemented in hardware.

The `HLS PIPELINE` directive put subsequent iterations of the corresponding loop into a pipeline, so that the execution of the next iteration starts as soon as the first instruction of the current one is completed. Since this directive is put on the loop that iterates over the output feature maps, when the pipeline is filled one pixel of the corresponding output channel is produced at each clock cycle.

In fact, the `UNROLL` directives in the inner loops instantiate the functional units to perform $Ch \times K_H \times K_W$ multiplication in parallel and then accumulating the results. Then, after one row of each output feature map is completed, the window is shifted up and the computation proceed to the next iteration.

The structure of the convolutional layer is able to produce one pixel on the output stream at each clock cycle, which is necessary to have a dataflow execution of the overall network.

4.3 Max-Pooling Layer Accelerator

The most used type of sub-sampling layer in many CNN architectures is the max-pooling layer. Since only the pixel of maximum values inside a kernel window is forwarded to later stages in the network, the max-pooling layer allows to save only the most relevant information for the classification process. Moreover, the sub-sampling of the pixel significantly reduces the dimensions of the resulting feature maps, thus saving a lot of memory.

The reduction of the memory also has a relevant effect on the amount of resources used to implement the accelerator on the FPGA. Indeed, as stated in the previous section, the dimension of window buffers is also determined by the width of the feature maps. Reducing the size of window buffers allows to save LUTs and possibly to implement larger models, even in terms of number of layer or compute units that operate in parallel.

The max-pooling accelerator module is implemented according to the code reported in Listing 4.3. As it can be seen, also in this case an `hls::Window` is used to store the portion of the pixels to get a row of the corresponding output feature map. However, since the stride of the pooling kernel is usually equal to the kernel dimension, it is necessary to perform enough shifts of the window until the proper number of new rows are loaded into the buffer.

As it happens for the convolutional layer, a `HLS PIPELINE` directive is put in the loop that iterates over the feature maps in order to write one pixel on the output FIFO at each cycle.

```
void maxPoolLayer (hls::stream<data_t> &streamIn, hls::stream<data_t> &streamOut){
    for (int i = 0; i < DIMH/PS; i++) {
        for (int j = 0; j < DIMW; j+=PS) {
            for (int fm = 0; fm < FM; fm++) {
#pragma HLS PIPELINE
                data_t max = -FLT_MAX;
                for (int ph = 0; ph < PH; ph++) {
                    for (int pw = 0; pw < PW; pw++) {
                        data_t pixel = window.next();
                        if (pixel > max)
                            max = pixel;
                    }
                }
                streamOut.write(max);
            }
        }
        for (int p = 0; p < PS; p++)
            window.shiftUp();
    }
}
```

Listing 4.3: C++ implementation of the max-pooling layer module.

4.4 Fully-connected Layer Accelerator

The fully-connected layer has a different computation pattern with respect to the convolutional and the max-pooling ones. Indeed, while those layers are based on kernels that perform the given operation on a small subset of the input pixels, fully-connected layers use all the input values to compute each element of the output vector. For this reason, it is not necessary in this case to insert a window buffer. In fact, each *i-th* pixel read from the input FIFO can be used directly by multiplying the weights $w_{i,j}$ and calculate each *j-th* output.

Listing 4.4 reports the implementation of the fully-connected module. The `HLS UNROLL` directive makes the output neurons to be computed in parallel. For bigger layer dimensions, the amount of computational units (i.e. *Digital Signal Processors* (DSPs) in case of floating point multiplication and addition) required to completely parallelize the inner loop may exceed the number of resources available on the programmable logic of the FPGA. However, it is always possible to instantiate only a portion of the output neurons to operate in parallel.

```
void fcLayer (hls::stream<data_t> &streamIn, hls::stream<data_tt> &streamOut,
    data_t weights[INPUT_NEURONS][OUT_NEURONS], data_t bias[OUT_NEURONS]){
    for (int i = 0; i < INPUT_NEURONS; i++) {
#pragma HLS PIPELINE
        data_t pixel = streamIn.read();
        for (int j = 0; j < OUT_NEURONS; j++) {
#pragma HLS UNROLL
            data_t w = weights[i][j];
            data_t tmp_mul = w * pixel;
            acc[j] += tmp_mul;
        }
    }

    for(int j = 0; j < OUT_NEURONS; j++){
#pragma HLS PIPELINE
        streamOut.write(acc[j] + bias[j]);
    }
}
```

Listing 4.4: C++ implementation of the fully-connected layer module.

This can be achieved by simply adding an *unrolling factor* in the directive. In this way it is possible to balance the tradeoff between performance and resource consumption of the desired implementation. Furthermore, the `HLS PIPELINE` di-

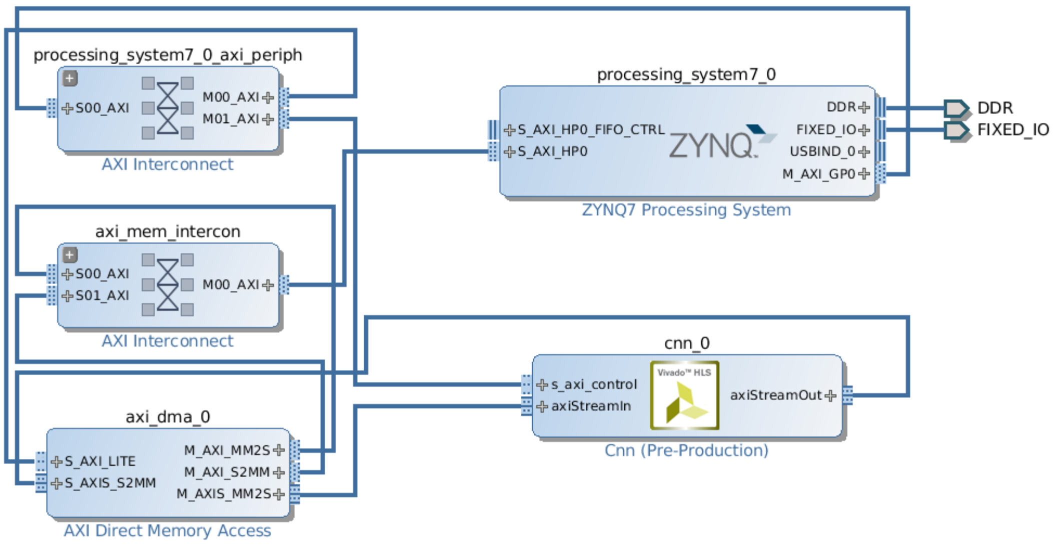

Figure 4.3: Zynq-7000 block design.

rectives allow to overlap subsequent execution of the loops, thus mantaining the dataflow property of the accelerator to obtain a higher throughput.

4.5 FPGA Hardware Design

Independently from the CNN accelerator, the FPGA needs to be configured with the modules for the run-time control and the memory management. The overall hardware design can be different according to the target device. Indeed, at the moment the framework supports three different platforms: the Zybo [22] and the Zedboard [23], both with powered by a chip of the Zynq®-7000 *All-Programmable System on Chip* (APSoC) family, and the VC707 Development Board [24], powered by a Virtex®-7 FPGA. Despite the limited number of devices, the design procedure for the two supported boards would be the same for other chips of the same family, thus the approach is easily extendable. Moreover, the same property holds also for the Virtex-7 and other FPGA-only devices.

4.5.1 Zynq Hardware Design

The Zynq-7000 devices are powered by a *System on Chip* (SoC) composed of an ARM processor and the Programmable Logic of the FPGA. The hardware design for these platforms exploits the ARM CPU in order to manage the run-time of the computation. Such design contains the ZYNQ7 Processing System (consisting of the hardwired ARM dual-core processor), an AXI *Direct Memory Access* (DMA) module, an AXI Interconnect and the CNN IP Core generated from Vivado HLS.

The framework allows also to instantiate more than one CNN core in the design, each with its own DMA module for data transfer. In this way is it possible to obtain a *coarse-grained* parallelism for the inference of the given dataset.

The resulting *Block Design* from Vivado is illustrated in Figure 4.3. The processing system takes advantage of the AXI High Performance slave interfaces (up to 4 ports) to transfer data to the DMA through the interconnect. In this way, data are directly moved between the accelerator and the on-chip main memory, without the need of CPU intervention. Moreover, each of the instantiated CNNs cores will be connected to a different DMA and, consequently, evenly distributed among the memory ports available on the target platform.

4.5.2 Virtex-7 Hardware Design

The hardware design for the Virtex-7 FPGA has a vey similar structure to the Zynq design. However, this type of device does not have an hard processor in the fabric, it is necessary to instantiate modules for the communication with the DDR memory and to manage the run-time. For this reasons, in such design a MicroBlaze soft-processor takes the place of the Zynq7 Processing System.

Furthermore, a *Memory Interface Generator* (MIG) IP Core is added to provide standard interface to the memory channels of the on-board DDR. The data transfer to the CNN core is done through a FIFO connected to a DMA module as for the hardware design for the Zynq devices. As it happens for the Zynq design, even for the Virtex-t it is possible to instantiate multiple groups of CNN and DMA modules by exploiting the four memory channels from the on-board DDR to the FPGA.

For sake of completeness, the block design of the Virtex-7 device is illustrated in Figure 4.4.

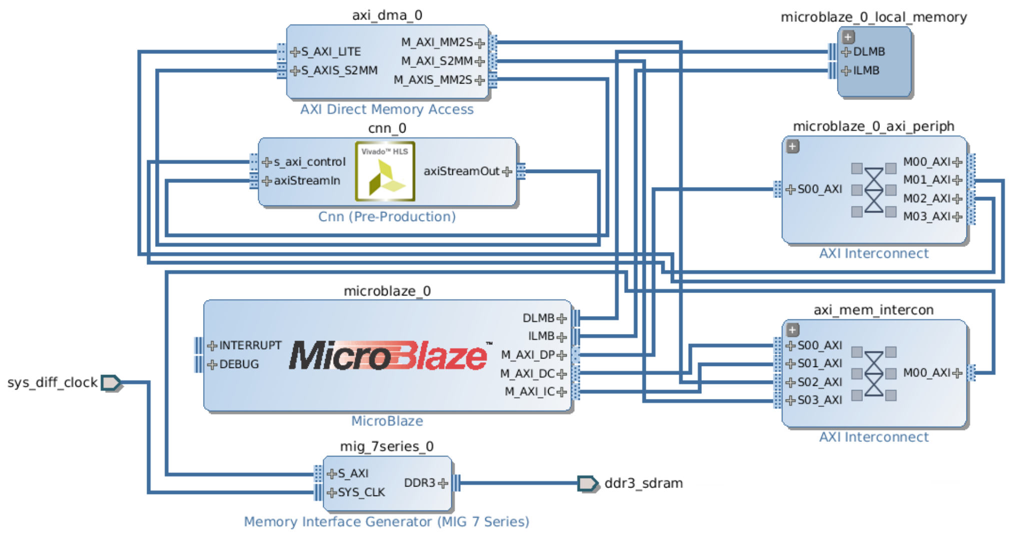

Figure 4.4: Virtex-7 block design.

Experimental Results 5

This chapter shows the experimental results of the validation of the proposed approach. Firstly, Section 5.1 describes the experimental setup in terms of case studies, target platforms and metrics. Then, the two following sections provides the details on the two different case studies. In particular, Section 5.2 describes the results obtained by the implementation of CNN accelerators for the USPS dataset, while Section 5.3 illustrates the outcome on the well-known MNIST dataset using two different network architectures. Finally, Section 5.4 provides a summary of the evaluated case studies, making also an analysis of the development time of such models using the framework.

5.1 Experimental Setup

The proposed framework has been developed as a set of Python modules to implement the different functionalities of the provided toolchain. In particular, the modules are written using Python 2.7 for legacy compatibility purpose. The module responsible for the training relies on the TensorFlow libraries by Google, while the internal representation structure is described with Google Protocol Buffers version 2. Then, the tools for the synthesis of accelerators are interfaced with the Xilinx Vivado HLx Editions 2016.2.

5.1.1 Target platforms

In order to validate the proposed approach, two different families of platforms have been used as target devices. As regards embedded-oriented devices, the Zynq APSoC family provides a variety of chips with limited resources available on the FPGA. Indeed, the selected Zynq target platforms is the Zedboard [23] board, which is powered by the Z7020 SoC, one of the smallest chips of the family. The SoC integrates also a dual core ARM Cortex-A9 that runs the PetaLinux Operating System.

Then, the VC707 Development Board [24] has been selected in order to evaluate the results for high-end devices. This board is powered by the Virtex-7, which has much more resources available on the programmable logic with respect to the aforementioned Zynq device. The control software on such device is run on the MicroBlaze soft processor, which boots in stand-alone mode.

Table 5.1 shows the amount of usable resources on the target FPGAs devices, pointing out the significantly different scale of the available area on the programmable logic.

Device	BRAM (18Kb)	DSP	FF	LUT
Zedboard (Zynq z7020)	280	220	106 400	53 200
VC707 (Virtex-7)	2 060	2 800	607 200	303 600

Table 5.1: Amount of resources available on the programmable logic in the two different FPGAs considered as target devices.

The performance of the proposed approach have been evaluated by considering different aspects. First of all, the acceleration gain obtained by offloading the inference on the FPGA is measured in terms of execution time and throughput in *Frames Per Second* (FPS). All the accelerators have been synthesized with a clock frequency of 100 MHz. As regards the energy consumption, we measured the power absorbed by the whole board using the *Energy Logger 4000* by Voltcraft [71]. Moreover, we provide the area utilization of the programmable logic of the Zynq APSoC and the Virtex-7 for the different network implementations in terms of the resources in Table 5.1.

5.1.2 Case studies

Since the publication of the work of LeCun et al. [11], one of the most analyzed case study for CNNs is the classification of handwritten digits. According to this tradition, this work analyzes four CNN architectures trained on two different datasets of handwritten digits, i.e. the USPS and the MNIST datasets. The models have different topologies in terms of number of layers and hyperparameters, leading to a different number of operations to process a single image. In particular, since the accelerators have been implemented by using 32-bit floating point datatype, the number of operations is expressed in *Floating-point Operations* (FLOPs).

Each of the test benches offloads the entire inference process of the two different datasets on the FPGA. The results are then collected and normailized by using a SoftMax operator from the host code.

5.2 Case Study: U.S. Postal Service dataset

The first case study of two different CNN models able to recognize the handwritten digits of the USPS dataset. This dataset is composed of 16x16 grey-scale images of digits scanned from the envelops of the U.S. Postal Service, and it was firstly utilized by LeCun et al. in [72].

The two CNN architectures, called *Small* and *Large*, have been trained with the TensorFlow process embedded in the framework. Then, the C++ code implementing the two networks has been generated starting from the ProtoBuf

Layer	K_{size}	K_{stride}	InFM	OFM	InDim	ODim	FLOPs
Conv1	5	1	1	6	16	12	44064
Pool1	2	2	6	6	12	6	864
FC1	–	–	–	–	216	10	4330

(a) Small USPS-Net

Layer	K_{size}	K_{stride}	InFM	OFM	InDim	ODim	FLOPs
Conv1	5	1	1	6	16	12	44064
Pool1	2	2	6	6	12	6	864
Conv2	5	1	6	16	6	2	19264
FC1	–	–	–	–	64	10	1290

(b) Large USPS-Net

Table 5.2: Architecture of *Small* and *Large* CNNs for USPS dataset recognition. The tables report the hyper-parameters of the layers and the number of floating point operations to process a single image.

represention of the models. As test bench for the two CNNs, we used 1000 images from the USPS test-set and compared the accuracy of the prediction and the performance with respect to a CPU multi-threaded execution.

Tables 5.2a and 5.2b report the structure of the two CNN models, while Table 5.3 shows the results in terms of performance, energy consumption and resource utilization of the different implementations. The execution time is compared to a multi-threaded (2 thread) software implementation running on the ARM Cortex-A9 of the Zedboard.

5.2.1 Small USPS-Net

The first test case has been performed on a CNN with a simple structure. This network is composed of a single convolutional layer, followed by a Max-pooling layer and a linear layer at the end. As it can be seen from Table 5.2a, the convolutional layer is made of six 5x5 kernels that take in input a 16x16 grey-scale image (single channel). The resulting six feature maps have dimensions 12x12;

(a) USPS Hardware vs. Software implementations

Test		Execution time		Speedup	FPS	Power		Energy		Test Error
CNN	Device	SW	HW			CPU	Device	SW	HW	
Small	Zedboard	1.67 s	0.028 s	59.64x	35714	2.2 W	4.40 W	3.67 J	0.12 J	3.7%
Small	VC707	1.67 s	0.017 s	98.23x	58823	2.2 W	20.01 W	3.67 J	0.34 J	3.7%
Small	VC707 (4 Cores)	1.67 s	0.0044 s	379.54x	227272	2.2 W	21.1 W	3.67 J	0.09 J	3.7%
Large	VC707	2.16 s	0.017 s	127.06x	58823	2.2 W	20.9 W	4.75 J	0.36 J	8.3%

(b) FPGA resources utilization

Test		Resources			
CNN	Target device	Flip-Flops	LUT	BRAM	DSP Slices
Small	Zedboard	31110 (29.24%)	32909 (61.86%)	11 (7.86%)	137 (62.27%)
Small	VC707	54777 (9.02%)	52238 (17.21%)	21.5 (2.09%)	143 (5.11%)
Small	VC707 (4 Cores)	162704 (26.80%)	145684 (47.99%)	66.5 (6.46%)	554 (19.79%)
Large	VC707	223843 (36.86%)	130433 (42.96%)	21 (2.04%)	895 (31.96%)

Table 5.3: USPS-Nets performance and resource utilization. Table 5.3a shows the results in terms of execution time, FPS and energy consumption of the two models. Table 5.3b reports the amount of resources used by the considered implementations on the different target devices.

then, the Max-pooling layer reduces the feature maps of a factor 2, resulting in a 6x6x6 cube. Each pixel of the cube becomes then an input neuron to the linear layer, composed of 10 neurons that predict the values corresponding to the 10 classes of digits to be recognized. Despite the simple structure, this CNN model is able to reach 96.3% of accuracy in the classification task, with only 37 mispredictions over the test set.

Three different configurations of hardware accelerators have been tested for the network to evaluate the gain in performance with respect to the software implementation. The first configuration consists in the hardware implementation on the Zedboard with single precision floating point. As it can be seen from Table 5.3a, this implementation is able to obtain a speed up in terms of execution time of 59.64x with respect to the multi-threaded software execution (2 threads), processing images at 35714 FPS. Moreover, thanks to the significantly reduced execution time, the obtained accelerator is more energy efficient than the software version running on the ARM processor (with an energy consumption of 3.67 J), even considering the whole System on Chip.

As regards resource utilization (Table 5.3b), the implementation of the *small* CNN model on the Zedboard makes an extensive use of DSPs and LUTs to implement floating-point adders and multipliers. Indeed, even though the fully-parallelized modules of the accelerator allow to obtain extremely low latency, the amount of resources to build such modules make impractical to implement large networks on resource-constrained devices. However, this problem can be handled by reducing the *unrolling factor*, thus implementing less processing elements per module, and *folding* the kernels computation in more iterations.

Finally, the *small* USPS CNN has been implemented also on the VC707 board. As it can be seen from Table 5.3a, the different lithography of the Virtex-7 FPGA is able to provide a faster implementation of the same accelerator, resulting in 0.017 s of execution time, and a speed up of 98.23x with respect to the pure software implementation, and almost 60 000 FPS. Despite the higher power drain of the Virtex-7, the energy required for the computation is still very small when compared to the ARM processor, with only 0.34 J of energy consumption.

Then, the greater number of available resources of this FPGA allowed also to implement a 4-cores configuration of the accelerator, obtaining a 4x faster implementation than the single core one and further reducing the energy consumption (0.09 J).

In terms of resource utilization, the Virtex-7 single-core implementation is similar to the one on the Zedboard. Indeed, the only differences that can be noticed are due to the presence of the MicroBlaze soft processor and the MIG as memory controller. The multi-core implementation uses almost 4x resources as expected, saving a portion of the area thanks to resource sharing.

5.2.2 Large USPS-Net

In this test case we evaluate the performance of the proposed approach when the dimensions of the CNN increases. In fact, this second model of network has an additional convolutional layer with respect to the previous one, thus augmenting considerably the *multiply-accumulate* operations that have to be performed during the image recognition process of the USPS test set. The additional convolutional layer takes as input the six 6x6 feature maps computed by the previous Max-pooling layer and it applies sixteen 5x5 kernels that produce the corresponding 2x2 feature maps. Then, the 16x2x2 is the input of the following linear layer, composed of 10 neurons associated to the classes as in the smaller CNN.

As reported in Table 5.3, the hardware implementation on the VC707 board of this larger CNN, performs slightly better than the previous ones in terms of speed-up; indeed, the enhanced number of *multiply-accumulate* increases the gap between CPU and FPGA. However, from Table 5.3b it can be noticed an increased amount of occupied area on the programmable logic. For this reason, it was not possible to implement the CNN hardware accelerator on the Zedboard due to the limited available resources. The implementation for the VC707 obtained a speedup of 127.06x compared to the ARM multi-threaded reference; furthermore, this design consumes only 0.36 J of energy.

An interesting note is that the prediction error of this network is worse than the previous one (8.3% against 3.7%), however this is not surprising because in the supervised learning context, too complex models can overfit the training set, thus resulting in a worse generalization capability on the testset.

5.3 Case Study: MNIST dataset

As a second case study, this work analyzes the results of FPGA-based accelerated inference of one of most popular datasets, i.e. the MNIST dataset. This datasets is composed of 28x28 black and white images of handwritten digits as illustrated in Figure 2.7 in Chapter ??. Although similar to the previous one, the bigger size of the input images significantly increases the amount of floating-point operations needed to process a single image, thus providing further useful results for the analysis of the performance of the accelerator generated by the framework.

As for the USPS case study, two different CNN architectues have been tested. The first one is a custom network topology named MNIST-Net. It is composed of two convolutional layers with max-pooling and one fully-connected layer. The second one instead is a variant of the model presented in [11] known as LeNet-5, composed of three convolutional layers and three fully-connected layers. Table 5.4 reports the dimensions of the layers of the two networks and the FLOPs for each layer to process a single image.

The two networks have been tested on the inference of the 10 000 images of the MNIST test set. The results are measured in terms of execution time, FPS and energy efficiency, also comparing with a pure software execution (Table 5.5a). The execution time is compared to a multi-threaded (4 threads) software implementation running on a Intel Core i7 6700HQ CPU. Moreover, Table 5.5b shows the resource utilization of the implemented accelerators on the FPGA.

5.3.1 MNIST-Net

This CNN topology is similar to the one of the *Large* USPS-Net. The convolutional part is composed of a pair of convolutional and max-pooling layers. The first convolutional layer has six 5x5 kernels that produce the same number of 24x24 feature maps. The subsequent sub-sampling layer halves each dimension of the features maps by applying 2x2 kernels with stride equal to 2. The same pattern is repeated in the next pair of convolutional and pooling layers. More precisely, the second convolutional layer is composed of sixteen 3x3 fil-

Layer	K_{size}	K_{stride}	InFM	OFM	InDim	ODim	FLOPs
Conv1	5	1	1	8	28	24	235008
Pool1	2	2	8	8	24	12	4608
Conv2	3	1	8	16	12	10	232000
Pool2	2	2	16	16	10	5	1600
FC1	–	–	–	–	400	10	8010

(a) MNIST-Net

Layer	K_{size}	K_{stride}	InFM	OFM	InDim	ODim	FLOPs
Conv1	5	1	1	6	28	24	176256
Pool1	2	2	6	6	24	12	3456
Conv2	5	1	6	16	12	8	308224
Pool2	2	2	16	16	8	4	1024
Conv3	4	1	16	64	4	1	32832
FC1	–	–	–	–	64	64	8256
FC2	–	–	–	–	64	32	4128
FC3	–	–	–	–	32	10	650

(b) Variant of LeNet-5

Table 5.4: Architecture of the two CNNs for inference of the MNIST dataset. The tables report the hyper-parameters of the layers and the number of floating point operations to process a single image.

ters, while the pooling has the same dimensions as the previous one. Then, the fully-connected layer perform the dot-product between the 400 inputs and the 10 neurons in output.

After the training process of the framework, this model is able to reach 97.82% accuracy. Although this is less than state-of-the-art accuracy, it still has a low prediction error and may be improved with small changes in the network topoplogy, such as adding ReLU activation functions to the neurons.

The size of the layers of this CNN results in an increased number of opera-

(a) MNIST Hardware vs. Software implementations

Test		Execution time		Speedup	FPS	Power		Energy		Test Error
CNN	Device	SW	HW			CPU	Device	SW	HW	
MNIST-Net	VC707	0.29 s	0.081 s	3.33x	123456	42.7 W	20 W	12.38 J	1.62 J	2.18%
LeNet-5	VC707	0.3 s	0.087 s	3.7x	114942	43.8 W	20.5 W	13.14 J	1.66 J	1.83%

(b) FPGA resources utilization

Test		Resources			
CNN	Target device	Flip-Flops	LUT	BRAM	DSP Slices
MNIST-Net	VC707	108995 (17.95%)	131469 (43.3%)	21 (2.04%)	510 (18.21%)
LeNet-5	VC707	258645 (42.6%)	267358 (88.06%)	67.5 (6.55%)	2296 (82%)

Table 5.5: Performance and resource utilization of networks for MNIST dataset inference. Table 5.5a shows the results in terms of execution time, FPS and energy consumption of the two models. Table 5.5b reports the amount of resources used by the considered implementations on the different target devices.

tions that reaches 0.48 *Mega Floating-point Operations* (MFLOPs) to process a single image. The implemented hardware accelerator is able to process the 10 000 images of the test set in 0.081 s, providing a throughput of 123456 FPS. Compared to the pure software execution, the hardware accelerated version achieves a 3.33x speed-up in terms of execution time. Although this may seem a modest improvement, by considering the *performance-per-watt*, the FPGA is able to obtain 6172.8 $^{FPS}/_{W}$, against 807.5 $^{FPS}/_{W}$ of the CPU.

From Table 5.5b it can be seen the increase in the amount of utilized resources with respect to the previous models. In particular, the size of the convolutional layers of the MNIST-Net results in an increased number of computational units for floating-point *multiply-accumulate* operations, i.e. DSPs and LUTs.

5.3.2 LeNet-5

The last case study consists in the analysis of the popular CNN model called LeNet-5. As stated before, the original model was first proposed by LeCun et al. [11]. In this work, a slightly different model has been generated by the framework and implemented on a Virtex-7 FPGA.

As illustrated in 5.4b, the CNN architecture is made by a total of eight layers: three convolutional layers, two max-pooling layers and three fully-connected layers. The first two convolutional layers have six and sixteen 5x5 kernels, respectively, and are followed by max-pooling layers with 2x2 filters and stride equal to 2. The last convolutional layer takes sixteen 4x4 feature maps as input and produces 64 single pixels in output by applyng 4x4 filters. Then, the fully-connected layers are composed of 64, 32 and 10 neurons, respectively.

In terms of accuracy, this model is slightly better than the previous one, providing 1.83% classification error, which is closer to the test error of the original *vanilla* implementation of LeNet-5 (0.95%).

Due to the number of layers and the filters dimensions, this network requires 0.54 MFLOPs to classify one image, resulting in 5.4 *Giga Floating-point Operations* (GFLOPs) for the inference of the entire test set. As it can be seen from Table 5.5, the CNN accelerator is able process the test set in 0.087 s, achieving a peak of almost 115 000 FPS. The execution time of the multi-threaded software version is instead 0.3 s. Considering these results, the *performance-per-watt* ratio for the two implementations are 5607 $^{FPS}/_{W}$ for the FPGA, and 761 $^{FPS}/_{W}$ for the CPU, providing a more efficient solution when the computation is offloaded in hardware.

As regards resources utilization, from Table 5.5b it can be seen that the hardware implementation of this accelerator makes an extensive use of the available LUTs and DSPs, due to the high level of parallelism in each layer module, and the array partioning of the internal buffers as explained in Chapter 4. For these reasons it would not be possible to implement larger networks due to the limited amount of resources. However, future versions of the framework will take into account this limitation and allow to specify the number of computational units to be instantiated for each layer, thus making a tradeoff between latency and resource consumption.

Network	# Lines	FLOPs	FPS	$^{GFLOPS}/_{W}$	$^{FPS}/_{W}$
Small USPS-Net	864	49K	58K	144M	2.9K
Large USPS-Net	1 583	65K	58K	183M	2.78K
Mnist-Net	2 300	0.48M	123K	2.96G	6.15K
LeNet-5	9 181	0.54M	115K	3.25G	5.61K

Table 5.6: Results summary of the evaluated CNN accelerators. The results are provided in terms of lines of code, number of FLOPs, throughput, *performance-per-watt* ratio both in terms of *Frames Per Second* (FPS) over Watt and *Giga Floating-point Operations per Second* (GFLOPS) over Watt.

5.4 Framework Evaluation Summary

The main purpose of the proposed framework is to provide tools that allow to rapidly develop a CNN hardware accelerator targeting FPGAs. As stated before, this process can be complex and time consuming. Indeed, the models that have been used as case studies count up to more than 9000 lines of C++ code that have to been synthesized by the Vivado toolchain to obtain the final implementation of the accelerators. Starting from scratch, it would be necessary to spend many hours to get the same result. By using the framework instead, the source code can be automatically generated in a few minutes, for instance providing a high level description of the network model in a `prototxt` file.

Table 5.6 provides a summary of the implemented models, specyfing the number of lines of code, and the obtained results in terms of number of operations, throughput and *performance-per-watt* ratio considering the entire test set.

Conclusions and Future Works 6

This chapter provides an overall analysis of this thesis. Section 6.1 gives an overview of the proposed work, summarizing the rationale, the methodology and the obtained results. Finally, Section 6.2 concludes the thesis by analyzing the problems of the proposed approach, proposing solutions to overcome such limitations and providing suggestions on future directions for this work.

6.1 Proposed Work Summary

Convolutional Neural Networks are playing an important role as a new emerging technology for computer vision. Thanks to their capability, they have become one of the most analyzed approaches in the last years for many application fields, including Big Data Analysis, mobile robot vision, video surveillance and so on. In such contexts, due to the huge amount of data to be processed or the need for real time execution, it is crucial to find techniques to speed up the computation. Moreover, the huge number of operations of CNNs makes it impractical to implement them on CPUs, so researchers and engineers have been looking for hardware accelerators able to provide both fast and energy efficient implementations. Among the accelerators proposed in the literature, FPGAs demonstrated to be very effective as flexible, low-power devices for hardware acceleration of CNNs.

However, the process to design and implement FPGA-based accelerators requires experience in hardware design and long development time. HLS softwares such as Vivado HLS provide tools that allow to build CNN accelerators using high-level languages such as C and C++. Nevertheless, some knowledge in hardware design is still required in order to obtain efficient implementations, while usually machine learning frameworks (e.g. TensorFlow, Caffe and so on) allow to use productivity-level language such as Python as programming language, thus becoming easily accessible for most of the software developers.

The aim of the work presented in this thesis is to bridge the gap between FPGA hardware design and software development, providing a framework able to design and implement a CNN accelerator on such platforms, starting from an high level description of the newtork. In particular, as described in Chapter 3, the proposed framework work-flow consists in:

- An external interface compatible with a subset of Caffe `prototxt` definition of a CNN to specify the network topology. Starting from this definition and other configurable parameters, such as the target device and the datasets for training and testing, the framework generates an Intermediate Representation based on a custom Google Protocol Buffer message structure.

- A configurable training process to generate the weights used by the final hardware accelerator for inference. The software implementation of the model is built on top of the internal ProtoBuf representation by using TensorFlow APIs to perform the actual training. The framework accepts different dataset formats from the user as input, including common image formats as `png`, and the IDX format as the one used by the MNIST dataset.

- A module for the automated generation of the C++ source code of the desired CNN model. The code implements a set of HLS-oriented, templatized C++ functions for the different types of layer of a CNN, described in Chapter 4. The resulting source code is then synthesized by using the Vivado Design Suite, and inserted in a hardware design template depending on the specified target device.

Chapter 5 provides the experimental results of the proposed approach, showing the performance of different CNN accelerators implemented on FPGAs. In particular, the obtained implementation has been evaluated on four different network models for the inference of two different datasets, i.e. the USPS and the MNIST.

The hardware design generated by the proposed framework demonstrated to be very efficient in terms of energy consumption and Frames Per Second, outperforming the pure software version running on CPU.

6.2 Future Works

Although efficient, the proposed framework still has some limitations. In particular, there are two different aspects that is worth to notice. On the one hand, at the moment the framework only supports a few types of layers to build the CNN model, thus limiting the degrees of freedom in the design space of such networks. On the other hand, the generated hardware implementation uses floating point operations as data type, which results in an extensive utilization of the programmable logic due to the implementations of floating point additions and multiplications with DSPs. Moreover, the generated architecture is massively parallel in terms of the number of computational units instantiated for each layer. For this reason it is not possible to implement very deep networks due to the limited resources available on the FPGA.

However, the modularity of the framework and the configurability of the hardware libraries allow scalability and flexibility to add more features and overcome this limitations. In fact, it is possible to further explore two different directions as future improvements for the proposed work:

- From the framework perspective, future extensions of this work will add other types of layer such as activations layer for ReLU, hyperbolic tangent and so on. Moreover, other degrees of freedom can be added to the training strategy in order to obtain better accuracy.

- From the hardware perspective, future architectures will use fixed-point data types instead of floating point, reducing also the precision in terms of bitwidth. Adopting low-precision data types would allow to significantly reduce the resource utilization on the target device, and exploits FPGA compute capability with few bits operands. Furthermore, new versions of the hardware library will include parameters to determine the number of compute units to be instantiated for each layer in the network. In this way it would be possible to properly tune the parallelism of the desired architecture, making a tradeoff between latency and resource utilization.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[2] Amazon.com, Inc. AWS Rekognition. Available online at `https://aws.amazon.com/rekognition/`.

[3] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[4] Apple Inc. Siri. `http://www.apple.com/ios/siri/`.

[5] Google Inc. Google Photos. Available online at `https://photos.google.com/`.

[6] Riccardo Petrolo, Valeria Loscrí, and Nathalie Mitton. Towards a smart city based on cloud of things. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, pages 61–66, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3036-7.

[7] P. A. Laplante and N. Laplante. The Internet of Things in Healthcare: Potential Applications and Challenges. *IT Professional*, 18(3):2–4, May 2016. ISSN 1520-9202.

[8] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. Dianne: Distributed artificial neural networks for the internet of things. In *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT*, M4IoT 2015, pages 19–24, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3731-1.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2010.05.010.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 05 2015. URL `http://dx.doi.org/10.1038/nature14539`.

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

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] Torch Framework. Available online at `http://torch.ch`.

[15] Google Tensor Processing Unit. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[16] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 7–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3838-7. doi: 10.1145/2820975.2820980. URL http://doi.acm.org/10.1145/2820975.2820980.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[18] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, 2013. doi: 10.1109/ICCD.2013.6657019.

[19] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In *Proc. of IEEE Int. Conf. on Application-specific Systems, Architectures and Processors ASAP)*, pages 53–60, 2009.

[20] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proc. of ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, 2015.

[21] Giulia Guidi, Lorenzo Di Tucci, and Marco D Santambrogio. Profax: A hardware acceleration of a protein folding algorithm. In *Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI), 2016 IEEE 2nd International Forum on*, pages 1–6. IEEE, 2016.

[22] Xilinx Inc. Zybo Zynq™-7000 Development Board. http://www.xilinx.com/products/boards-and-kits/1-4azfte.html, .

[23] Zedboard. http://zedboard.org/product/zedboard.

[24] Xilinx Inc. Virtex-7 FPGAs. https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html, .

[25] Altera (Intel). Stratix-V FPGA. https://www.altera.com/products/fpga/stratix-series/stratix-v/overview.html.

[26] Project Jupyter. Available online at https://jupyter.org/.

[27] Xilinx Inc. Vivado Design Suite. http://www.xilinx.com/products/design-tools/vivado.html, .

[28] MNIST handwritten digits dataset. Available online at http://yann.lecun.com/exdb/mnist/.

[29] CIFAR-10 and Cifar-100 datasets. Available online at https://www.cs.toronto.edu/~kriz/cifar.html.

[30] Frank Rosenblatt. The perceptron a perceiving and recognizing automaton. Technical report, tech. rep., Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957. 2, 1957.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, (323):533–536, 1986.

[32] Michael Nielsen. *Neural Networks and Deep Learning*. URL http://neuralnetworksanddeeplearning.com/.

[33] Anil Kumar Goswami, Shalini Gakhar, and Harneet Kaur. Automatic object recognition from satellite images using artificial neural network. *International Journal of Computer Applications*, 95(10), 2014.

[34] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1): 215–243, 1968.

[35] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[36] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[38] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[40] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 253–256. IEEE, 2010.

[41] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962. Citeseer, 2003.

[42] Kumar Chellapilla, Michael Shilman, and Patrice Simard. Optimally combining a cascade of classifiers. In *Proceedings of SPIE*, volume 6067, pages 207–214, 2006.

[43] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[44] Kumar Chellapilla and Patrice Simard. A new radical based approach to offline handwritten east-asian character recognition. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[45] Ahmad AbdulKader. A two-tier arabic offline handwriting recognition based on conditional joining rules. In *Arabic and Chinese Handwriting Recognition*, pages 70–81. Springer, 2008.

[46] Régis Vaillant, Christophe Monrocq, and Yann Le Cun. Original approach for the localisation of objects in images. *IEE Proceedings-Vision, Image and Signal Processing*, 141(4):245–250, 1994.

[47] Christophe Garcia and Manolis Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 26(11):1408–1423, 2004.

[48] Margarita Osadchy, Yann Le Cun, and Matthew L Miller. Synergistic face detection and pose estimation with energy-based models. *Journal of Machine Learning Research*, 8(May):1197–1215, 2007.

[49] Fabian Nasse, Christian Thurau, and Gernot A Fink. Face detection using gpu-based convolutional neural networks. In *International Conference on Computer Analysis of Images and Patterns*, pages 83–90. Springer, 2009.

[50] A Frome, G Cheung, A Abdulkader, M Zennaro, B Wu, A Bissacco, H Adam, H Neven, and L Vincent. Large-scale privacy protection in street-level imagery. *ICCV'09*, 1(2), 2009.

[51] Steven J Nowlan and John C Platt. A convolutional neural network hand tracker. *Advances in Neural Information Processing Systems*, pages 901–908, 1995.

[52] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3D convolutional neural networks for human action recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 35(1):221–231, 2013.

[53] Manolis Delakis and Christophe Garcia. text detection with convolutional neural networks. In *VISAPP (2)*, pages 290–294, 2008.

[54] Yann LeCun, Urs Muller, Jan Ben, Eric Cosatto, and Beat Flepp. Off-road obstacle avoidance through end-to-end learning. In *NIPS*, pages 739–746, 2005.

[55] Ronan Collobert and Jason Weston. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proc. of ACM Int. Conf. on Machine Learning (ICML)*, pages 160–167, 2008.

[56] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[57] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[58] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[59] D. Strigl, K. Kofler, and S. Podlipnig. Performance and Scalability of GPU-Based Convolutional Neural Networks. In *Proc. of Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 317–324, 2010.

[60] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proc. of Int. Joint Conf. on Artificial Intelligence*, volume 22, page 1237, 2011.

[61] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proc. of IEEE Int. Symp. on Circuits and Systems (ISCAS)*, pages 257–260, 2010.

[62] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, Oct 2013. doi: 10.1109/ICCD.2013.6657019.

[63] Maurice Peemen, Bart Mesman, and Henk Corporaal. Inter-tile Reuse Optimization Applied to Bandwidth Constrained Embedded Accelerators. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 169–174, 2015.

[64] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 32–37, 2009.

[65] Yongming Shen, Michael Ferdman, and Peter Milder. Overcoming Resource Underutilization in Spatial CNN Accelerators. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2016.

[66] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proc.of ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[67] Kruttidipta Samal. Fpga acceleration of cnn training. Technical report, 2015.

[68] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[69] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 32–37, 2009.

[70] Protocol Buffers. Available online at `https://developers.google.com/protocol-buffers/`.

[71] Voltcraft. http://www.voltcraft.com.

[72] Y LeCun, B Boser, JS Denker, D Henderson, RE Howard, W Hubbard, and LD Jackel. Handwritten digit recognition with a back-propagation network. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, pages 396–404. MIT Press, 1989.