Politecnico di Torino

Joint Master's Degree in Nanotechnologies for ICTs

In partnership with:
Grenoble INP-Phelma
École Polytechnique Fédérale de Lausanne

Implementation of a Face Detection Algorithm for Low-Power FPGAs

Supervisor:
Prof. Guido MASERA

Candidate:
Giovanni CAPPAI

External Supervisors:
Prof. Luca BENINI
Dr. Michele MAGNO

External partner university:

ETH zürich

Academic year 2019/2020

Acknowledgements

I would like to thank the Integrated System Laboratory of ETH, in particular Prof. Luca Benini and Dr. Michele Magno for giving me the opportunity of working on such an ambitious project that really helped my personal and professional growth. I would like to express my gratitude to Mr. Morirtz Scherer, for his precious help and patience in always answering my questions during the work on the project. I would also like to extend my gratitude to Prof. Guido Masera, for his availability and important feedback provided.

To my parents, Emanuela e Michele, for their endless support during all these years of studying abroad.
To all the friends who shared these last years with me, for the enjoyable moments that made me feel at home and, to my childhood friends, for still being with me despite the distance.

Torino, October 23, 2020 G.C.

Abstract

The recent technological progress is pushing the scientific community to enter a new Artificial Intelligence (AI) era, with systems able to perceive and understand the real world, in order to solve problems in a smarter way than humans do. In this field, Computer Vision (CV) systems are able to automatically see, identify, and understand the external visual world, making decisions depending on the data acquired, emulating the human vision. This progress is placed side by side with a continuous search of hardware architectures, such as GPUs, FPGAs, and ASICs, that can sustain the computation in a more efficient way than conventional software implementations do, allowing the analysis of data directly where it is acquired.

The project wants to address both tasks with the implementation of a face detection algorithm, with a Deep Learning approach, on a low-power FPGA, taking advantage of its full programmability. While different building blocks have already been demonstrated as independent proofs-of-concept, a fully integrated, stand-alone system has not been developed and the project tries to fill this hole.

This thesis presents the steps performed in order to obtain a suitable face detection algorithm that can fit within a limited memory, low-power FPGA, including pre- and post-processing of data. The work can be essentially divided into two parts: firstly a software implementation of the face detector has been done, and the results are compared to the literature, then simplifications are applied to the model in order to reduce its complexity for the successive hardware implementation on the target FPGA.

Contents

List of Tables

List of Figures

List of Acronyms

AI	Artificial Intelligence
CMOS	Complementary MOS
CNN	Convolutional Neural Network
CV	Computer Vision
DDR	Double Data Rate
DSP	Digital Signal Processing
EBR	Embedded Block RAM
EVDK	Embedded Vision Development Kit
FC	Fully Connected
FPS	Frame Per Second
HDL	Hardware Description Language
IoU	Intersection over Union

IP	Intellectual Property
LMMI	Lattice Memory Mapped Interface
MAC	Multiply-Accumulate
MAE	Mean Absolute Error
MMAC	Multiply-Multiply-Accumulate
MTCNN	Multitask Cascaded Neural Network
NMS	Non Maximum Suppresion
NN	Neural Networks
O-Net	Output Network
P-Net	Proposal Network
PR	Precision-Recall
PReLU	Parametric ReLU
R-Net	Refine Network
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent

Chapter 1

Introduction

In the last years, the use of Artificial Intelligence (AI) has been constantly increasing in every field of science due to the large amount of data to be processed. One of the most inspiring tasks for AI algorithms is the problem of making a digital system capable to see the external world and to make decision depending on what it sees, as humans commonly do with their sight. All the applications related to the analysis of the content of images or video fall into the name of Computer Vision (CV) applications. Almost every system that includes some camera it is also equipped with a CV algorithm, starting from the common face tracking of smartphones, to object detection algorithms for autonomous cars or face recognition for security systems.

Typically, systems that include an AI algorithm have a cloud structure, as represented in the next figure 1.1.

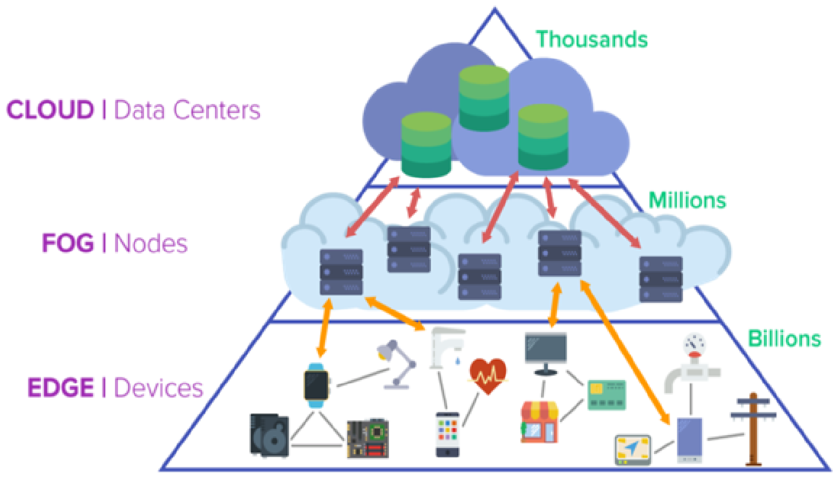


Figure 1.1. A typical cloud architecture.

The sensing devices, represented at the bottom of the pyramid, collect the external data that is sent to the cloud data centres for analysis and computation with the sensed data. At the end, depending on the task addressed, the results are sent back to the sensing device, in order to give to users a response.

The main advantage of this structure is the possibility to collect large amounts of data of different type and to perform complex operations thanks to the high computational power of data centres. On the other hand, it has the drawback of a long transmission latency due to communication needed between the bottom and the top of the pyramid. In recent years, the main trend is to move the computation directly where the data is acquired, integrating AI algorithms on edge devices (EdgeAI). This processing paradigm leads to a reduction of the transmission latency and protects the user privacy but introduces some other challenges such as reduced computational power and limited resources, due to the fact that edge devices are usually portable and tend to be less computationally powerful.
Edge devices can be classified depending on the structure on which they are based, i.e. there are CPU based devices with good versatility but with limited parallelism for computation, or GPU based devices whose strength is the massive parallelism but with high power consumption. Particularly attractive are FPGA-based devices, that allow the development of specifically customized designs with low latency and reduced power consumption. The main disadvantage is the limited memory available, that introduces challenges in the development of AI algorithms for devices with constrained memory.

1.1 The Project

Building on the recent trends in the fields of CV and EdgeAI applications, this project aims to address both tasks, with the implementation of a face detection algorithm for a low-power FPGA.

The problem of face detection has been chosen due to the fact that many CV tasks have as a primary objective the identification and localization of faces in images. Some of the most popular applications are face recognition and face tracking. Ever since the seminal work of [13], deep learning and Convolutional Neural Network (CNN) constitute the state-of-the-art approach for AI algorithms for CV, thanks to their capability of simplifying the feature extraction from input images, for the subsequent recognition of its content [4].
In addition, convolution operation is a relatively easy operation to be translated on hardware and its integration is a well-established practice, see for example works [2] and [12], and many other hardware implementations that can be found in literature. In fact, as explained in next section 2.5.1, the convolution operation can be simply

reduced to a weighted sum of elements, translated in hardware in the implementation of Multiply-Accumulate (MAC) operations. Usually, only the computation of the CNN is developed on FPGAs, in order to take advantage of the customizable design to obtain a highly parallel architecture for convolution operations, leaving the pre- and post-processing of data on external CPUs or GPUs. The novelty of the project is in the full integration of the detector for a completely low-power analysis of input data. On the other hand, it introduces some other challenges, because operations that are commonly done in a simple way on CPU or GPU, like rescaling of input images and post-processing of the data coming form the CNN, have to be changed and simplified due to the reduced computational power of an FPGA. The target board is the Embedded Vision Development Kit (EVDK) from Lattice Semiconductors™, that includes all the peripherals for acquiring and visualizing external data [27].

1.1.1 Goals of the project

The main goals for this experience can be synthesized in:

1. Comparison of existing solutions of face detection algorithms using a Deep Learning approach, in order to find a suitable network for the desired electronic application.

2. Training and evaluation of the chosen network.

3. Implementation with a FPGA-based accelerator. Comparison between the software and hardware solutions.

1.2 Outline

The thesis is organised as in the following:

- **Chapter 2** treats the theoretical background about Neural Networks (NN) and CNNs, essential for a complete understanding of the following chapters. The concepts of artificial neurons and neural network are introduced and the learning process of such networks is described. More focus is then given to CNNs, that is the method used for the work done.

- **Chapter 3** presents the hardware setup and its characteristics.

- **Chapter 4** discusses the software implementation of the face detector. Firstly the structure of the chosen CNN is analysed and its training is described. Secondly, modifications that have been made are discussed.

- **Chapter 5** deals with the FPGA implementation of the face detector. The chapter discusses how different operations have been translated from software to hardware. The second part shows the resource and timing results of this implementation.
- **Chapter 6** summarizes the results obtained and proposed some future steps for the project.

Chapter 2

Brief Introduction on Neural Networks Fundamentals

The aim of this chapter is to give to the reader a basic knowledge about the theory behind the project, in order to completely understand the following chapters. In particular, after an introduction to AI and Artificial NNs, without going very deep into mathematical theory, more focus is given to Deep Learning and CNNs that are at the base of the work done in this thesis.

2.1 Neural Networks

Artificial Neural Networks take their inspiration from the structure of the human brain and from the observation that it works in a extremely different way from a conventional digital computer. In fact, the brain can be seen as a *complex, non-linear and parallel* system, that in some cases, can perform computations many times faster than an existing digital system, thanks to its ability to reorganize its structural components, the neurons [8]:

> "A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:
>
> 1. Knowledge is acquired by the network from its environment through a learning process.
> 2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge."

In conclusion, NNs due their computing power to the parallelize structure and the *generalization*, the capability to produce consistent outputs given new inputs outside the training dataset.

2.2 Model of a Neuron

As said before, the objective of a NN is to, at least, try to mimic the human brain behaviour. Therefore, the first step is to model the basic building blocks, the *neurons*.
A schematic representation on how it is done can be seen in the next fig.2.1.

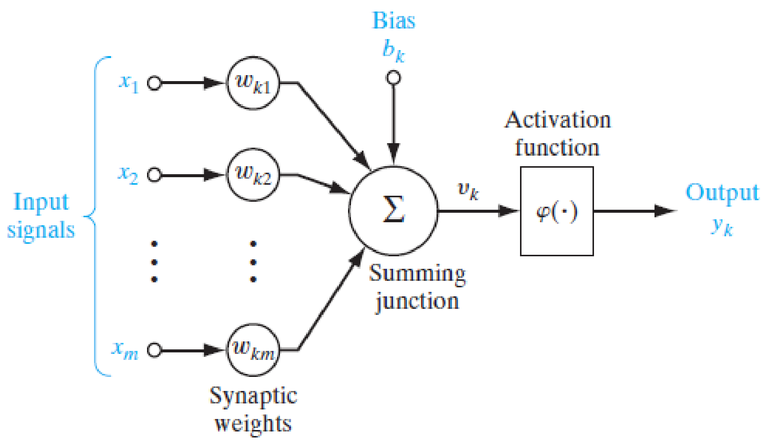


Figure 2.1. Schematic model of an Artificial Neuron. Adapted from [8].

Here, three fundamental elements can be found:

- A set of *synapses*, each of them takes an input signal and multiplies it by a constant called *synaptic weight*, specific for each synapse. As explained later, the final goal of the learning process is to determine these weights for the NN.
- An adder in order to sum all the weighted inputs.
- An activation function, foremost a non-liner one, is applied to generate the final output. Typical ranges for the output are $[0, 1]$ or $[-1, 1]$.

In a mathematical form, the model can be represented with the following two equations:

$$\nu_k = \sum_{j=1}^{m} x_j w_{kj} + b_k \tag{2.1}$$

and

$$y_k = \varphi(\nu_k) \tag{2.2}$$

where x_j is the input signal, w_{kj} is the weight for the j^{th} input of the neuron k, ν_k is called *activation potential*, $\varphi(\cdot)$ is the activation function and y_k is the output. b_k is called *bias*, is an external parameter with the function of increasing (lowering) the positive (negative) input of the activation function. Essentially it applies an *affine transformation*.

2.2.1 Types of Activation Functions

The most common activation functions are [20]:

1. Heaviside function.

$$y_k = \begin{cases} 1 & \text{if } \nu_k \geq 0 \\ 0 & \text{if } \nu_k < 0 \end{cases} \tag{2.3}$$

2. Sigmoid function. It is one of the most popular function used in NN, unlike the Heaviside function it is continuous and differentiable:

$$\varphi(\nu_k) = \frac{1}{1 + e^{-\alpha \nu_k}} \tag{2.4}$$

   where α is the slope factor.

3. Rectified Linear Unit (ReLU). Introduced by Hahnloser et al. [7], it outperforms the two previous activation functions, making it the most used function in the framework of Deep Learning [15]. Its general form is:

$$f(x) = \max(0, x) \tag{2.5}$$

4. Variations of the ReLU. Examples are the *Leaky ReLU*, in which for negative values of x it is defined as a linear function with constant slope, and the Parametric ReLU (PReLU) in which the slope for the negative values of x is a learning parameter.

2.3 Network Architectures

In order to have a working system, neurons must be arranged to form more complex structures. The basic architecture is the *Multilayer Feedforward Network* [8] where the neurons, also called *nodes*, are organized in *layers*.

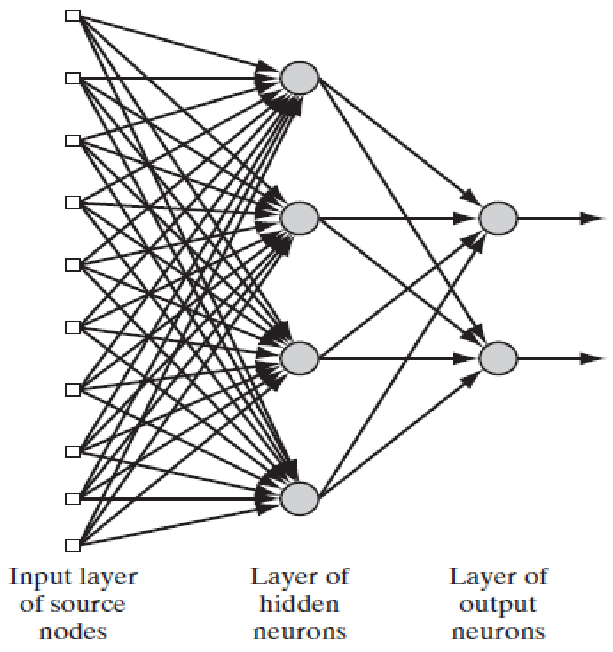


Figure 2.2. Fully connected Multilayer Feedforward Network. Adapted from [8].

Basically, there are three main components:

- An input layer of source nodes, that supply input data for successive layers and where no computation is done.
- One or more hidden layers, where most of the computation is performed. Their components are called hidden neurons (or units), due to the fact that they are not directly accessible from neither the input or output of the network. Multiple hidden layers can be stack one over the other in order to extract high-order information (called *features* in the framework of image classification). This idea is at the base of Deep Learning.
- Output layer, that gives the final response of the network for the given primary input.

In particular, the network reported in fig.2.2 is called *fully connected* because each node of each layer is connected to every node of its adjacent forward layer.

2.4 Learning Process

A *Learning Process* is a process through which the NN is trained for a particular task in order to then infer correct results when new unknown data is given at its input. Two main kind of processes can be distinguished: *unsupervised* and *supervised* learning. The focus is given to the latter, that is the most common strategy and the one used during the project.

2.4.1 Supervised Learning

Supervised Learning, also called *Learning with a teacher* [8], is the process where the network acquires knowledge of the external environment thanks to a feedback loop that compares the response of the network with the correct one, and tries to minimize a measure (loss function) of the error.
This is represented in the following fig.2.3.

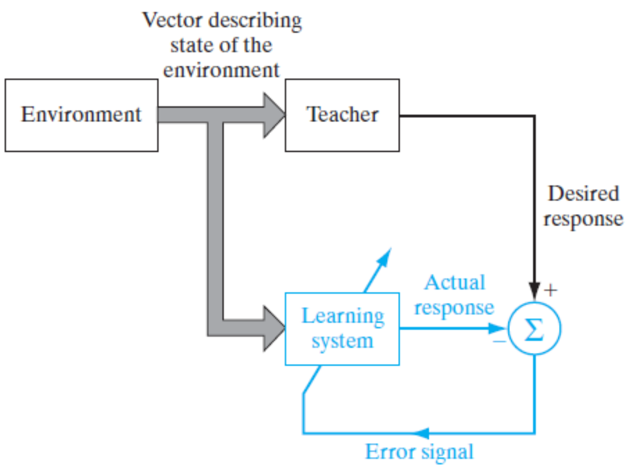


Figure 2.3. Schematic of a Supervised Learning Process. Adapted from [8].

During the process, the network parameters, the synaptic weights, are adjusted iteratively, step by step. The error signal is defined as the difference between the actual response from the network and the expected response. This error is represented through a *loss function* of the training parameters, differently defined depending on the task for which the NN is trained for. At the end of the training process, what the system has learned is represented in the fixed values for the weights and the network is then able to predict correct results with samples outside of the training dataset.
From a mathematical point of view, the loss function can be visualized as a surface

on a multidimensional space, where each axis is a training parameter [8]. The aim of the training process is to minimize the error, that is translated in trying to reach a minimum point on the multidimensional error surface. This is achieved by *backpropagation*: it uses the information from the gradient of the loss and specific optimization algorithms to minimize the loss function. The training process usually stops when some stopping criteria is achieved or a maximum number of iterations is reached.

2.4.2 Optimization Algorithms

The learning process can be expressed as an *optimization problem*, that is trying to minimize the loss function, and finding the values of weights that lead to this minimum. Using a mathematical formalism, it can be expressed as:

$$x^\star = \arg\min_{x*\in x} f(x) \tag{2.6}$$

where, in the framework of NN, $f(x)$ will be the loss function and $x^\star$ the final weights after training process, that generate the minimum loss.
In next sections, a brief description of Gradient Descent and its evolution Stochastic Gradient Descent (SGD), one of the most used optimization methods and the one used in the project, are presented.

Gradient Descent

The concept behind the gradient descent method is very simple, in order to reach the minimum of a function the information about the derivate of the function is used. In particular, if the derivate is negative it means that the slope of the function is also negative. By remembering the meaning of the derivative, at the first order it is possible to write:

$$f(x+\epsilon) \approx f(x) + \epsilon f'(x) \tag{2.7}$$

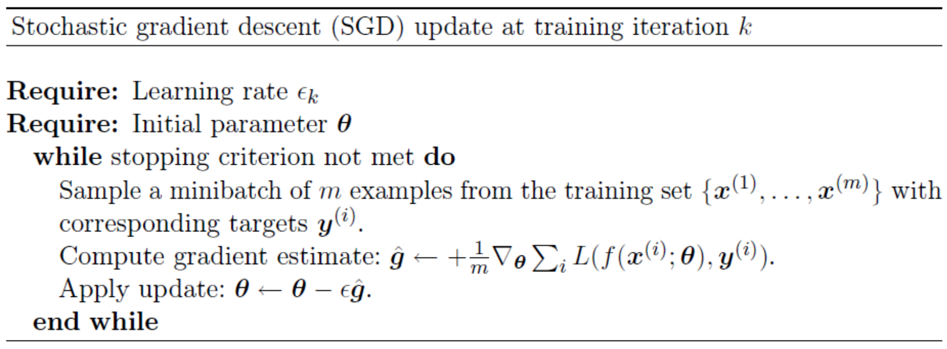
So the derivative is useful because knowing the sign of $f'(x)$ it is possible to know how to modify x in order to decrease the value of $f(x)$ [6]. This is exactly the information needed during the learning process, where the values of the weights are adjusted to obtain a decrease of the loss function.
This reasoning can be generalized in the case of multiple inputs by using the gradient instead of the derivative, and the function is decreased by moving in the direction of the negative gradient.
One single step of optimization can be written as:

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{2.8}$$

where **x'** is the corrected value of the input vector, $\mathbf{x}$ its previous value, $\nabla_{\mathbf{x}} f(\mathbf{x})$ is the gradient evaluated in $\mathbf{x}$ and ϵ is called *learning rate* and it is the size of the step.

Stochastic Gradient Descent

Considering the previous argument, one problem arises when dealing with its application to problems with a large training dataset. In these cases, the gradient used for the parameter's update is the average of the gradients calculated for each single input data. Having a dataset of m samples, the computational cost of each parameter's update is $\mathcal{O}(m)$ that is not feasible for large training datasets. SGD instead, at each update, approximate the gradient by calculating it over just a *batch* of data, sampled uniformly from the training dataset. In this way, the computation time for the single update doesn't increase with a growing number of training samples [6]. The complete algorithm of SGD is reported in the next fig. 2.4.



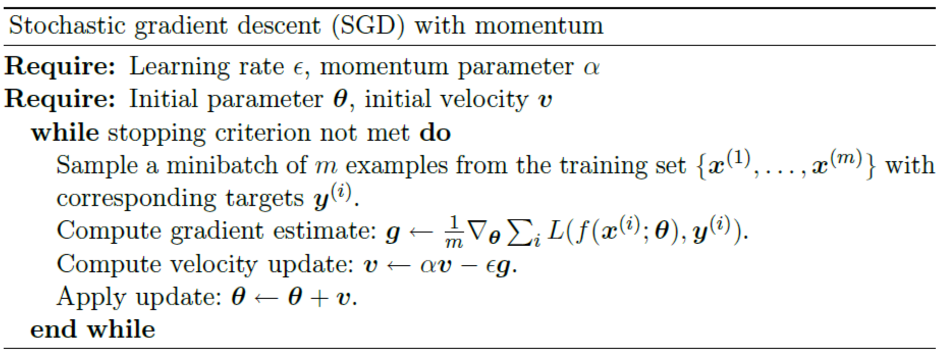
```
Stochastic gradient descent (SGD) update at training iteration $k$

Require: Learning rate $\epsilon_k$
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$.
  end while
```

Figure 2.4. SGD algorithm. Adapted from [6].

Usually training loss is not plotted versus steps but versus *epochs*, that is the number of steps needed to have a complete pass over the entire training dataset.

Momentum

In some cases, also the SGD algorithm can be too slow for some practical applications. For this reason, accelerator methods have been designed to improve the speed of the training process with SGD: one of them is the *momentum* [19].
With the addition of the momentum, the algorithm is modified with an exponentially decaying average of the previous gradients, in this way, larger steps are taken in the direction where the past gradients pointed. By doing that, if there is a sequence of gradients pointing in the same direction, larger steps are taken in this direction to speed up the reaching of the loss function's minimum [6]. The new algorithm is:



```
Stochastic gradient descent (SGD) with momentum
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

Figure 2.5. SGD with momentum algorithm. Adapted from [6].

The parameter $\alpha \in [0,1]$ defines the contribution of the momentum in the parameters' update. Higher is the value of α with respect to the learning rate ϵ, more the past gradients affect the direction of the updating step. Typically its value is near 1.

Hyperparameters

An *hyperparameter* is a parameter that must be set by the user to perform the training process in a correct way. Examples of hyperparameters are:

- Learning rate ϵ. Probably the most important one. It defines the size of the steps taken in the direction for minimizing the loss. Too large values can lead to an oscillating behaviour, because the minimum is overtaken by one of the steps, too small values decrease the velocity of the process. One usual technique is to adopt varying values of ϵ during training, starting from large ones and then moving to small ones when approaching the minimum.

- Batch size. As said, it is the number of samples used for one update of the parameters' values. Usually it is not a critical parameter, typically values are multiples of 2, i.e. 64, 128 or 256.

- α. It defines the contribution of the momentum on weights' update. Also in this case, a varying values solution can be adopted during training, in a similar manner as for learning rate.

Convex vs non-convex problems

In principle, all the discussion done about optimization algorithms is valid only for *convex* problems, such that all the local minima are also global minima. In these cases, the application of an optimization algorithm allows the reaching of the minimum. The important thing to be pointed out is that almost all the problems with NNs are *non-convex* ones. The application of optimization strategies in this situation does not ensure reaching even a local minimum. Anyway, has been observed that the results obtained are good enough for practical application with non-convex problems [6].

Overfitting

In some cases, can happen that the accuracy on the training dataset is quite high while the network fails when unknown data is given in input. This is a common problem called *overfitting*, the network models too well the training data and it is not able to generalize with new data. In order to avoid this behaviour, usually during learning process a *validation* dataset is also used. It is composed by images not used during training steps, at the end of each epoch the network is evaluated over this dataset in order to monitor the results with unknown data. If the validation loss is not decreasing over epochs, common strategies are the calibration of the hyperparameters or the change of the training dataset.
Another technique commonly used for reducing overfitting is the addition of a *regularization* term in the loss function during training.

Regularization

When performing the training process, what one wants to get at the end is a model that can work on new data, so a model that can generalize well. During the learning process, the weights are constantly updated, step by step, but no information on how the weights actually look is given. In fact, given a certain accuracy target, different weights matrices, that leads to this particular value of accuracy, can exist. The goal of regularization is to try to find a set of weights that at the end of training generate the most possible generalized model. It is done by a penalization of the largest weights' values on the weight matrix, because they influence more the output of the network and have more probability to lead to overfitting the training data [14]. The goal of regularization is to find lower values for these weights, that generate models with a better capacity to generalize. Common penalty functions are L1 and L2 norms:

$$R(\mathbf{W}) = \sum_i \sum_j |W_{i,j}| \tag{2.9}$$

$$R(\mathbf{W}) = \sum_i \sum_j W_{i,j}^2 \tag{2.10}$$

where $W_{i,j}$ is one weight of the matrix.
When defining the total loss, the general formula becomes:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(\mathbf{W}) \tag{2.11}$$

where λ is the hyperparameter that controls the strength of the penalization.
When using regularization, it is possible to see a slight increase of the training loss, but it leads to better accuracy on testing data.

2.5 Convolutional Neural Networks

CNNs are fundamentals for building Deep Learning applications for computer vision problems, because they are especially designed for multi-dimensional inputs, like standard RGB images. Differently from what happens in traditional feedforward networks (sec. 2.3), where each neuron is connected to every neuron of the next layer, in a CNN, Fully Connected (FC) layers are usually used just at the very end of the network, to generate the output depending on the problem addressed by the specific computer vision application.
In a CNN, each layer applies a different set of filters in order to extract, moving deeper inside the network, higher-level features to make predictions about the content of the input image. In order to do that, a large number of *convolutional* layers is needed and for this reason, by definition, a CNN is a type of Deep Learning algorithm [20]. During training, the values of the filters are the training parameters automatically learned by the network. The next figure 2.6 compares and summarizes the two different structures of a FC network and a CNN.
In conclusion, a CNN first detects edges from the input pixels, uses them to detect shapes and finally detects high level features like facial or object parts.

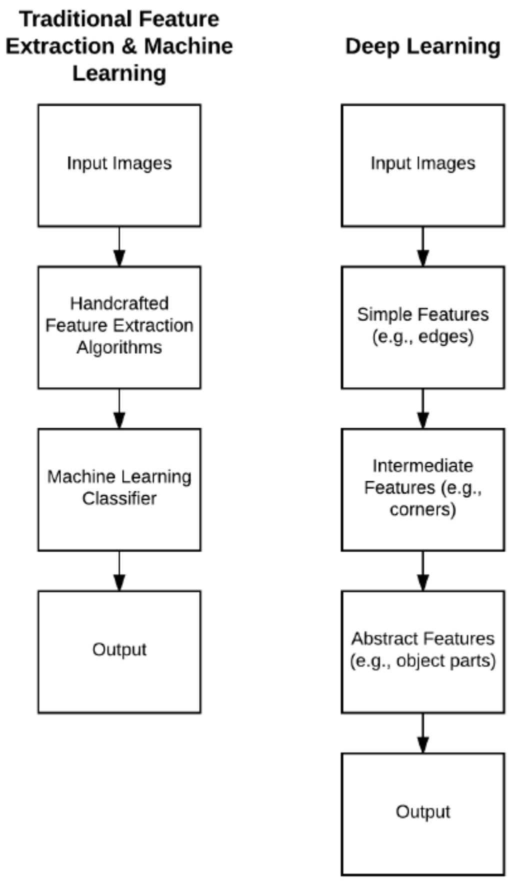

Traditional Feature Extraction & Machine Learning
Input Images
Handcrafted Feature Extraction Algorithms
Machine Learning Classifier
Output
Deep Learning
Input Images
Simple Features (e.g., edges)
Intermediate Features (e.g., corners)
Abstract Features (e.g., object parts)
Output

Figure 2.6. Comparison between a classical Machine Learning and Deep Learning approach for image classification. Adapted from [20].

2.5.1 Convolution Operation

Convolution is a well-known mathematical operation consisting in the integral of the product between one function, let's say $x(\tau)$, and another function translated by a fixed value, $w(t-\tau)$. In formulas:

$$(x \star w)(t) = \int x(\tau)w(t-\tau)d\tau \tag{2.12}$$

where the first argument is called *input*, the second one is called *kernel* and the result is the *feature map*.
Previous equation 2.12 is written in the case of continuous variables, but it can be also defined in the case of discrete ones. Also, nothing that in the framework of CV applications the analysis is done with images, the input and the kernel are represented with multi-dimensional arrays. In this case, that is the one important

for the aim of the project, the convolution operation can be written as:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.13}$$

or equivalently, due to the commutative property as:

$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.14}$$

where I is the input tensor and K is the kernel tensor.
It is important to point out that the commutative property can be applied because the kernel is flipped with respect to the input image [6]. Almost all of CNNs don't use the convolution operation but the *cross-correlation* one:

$$S(i,j) = (K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.15}$$

where there is not the flipping of the kernel matrix. In the framework of Deep Learning the terms convolution and cross-correlation are interchangeable and in the following this convention is used.
While its mathematical form could seem to be a difficult operation, it is not true in practice: the convolution operation can be simply thought as an element wise multiplication between two matrices, followed by a sum [20]. With this idea in mind, the convolution operation applied to an image can be visualized in the following terms:

- The RGB input image is a tensor with dimensions (h, w, c) where h is the height of the image (the number of pixel's rows), w is the width (the number of pixel's columns) and c represent the three channels of the image, red, green and blue.
- The kernel applied to the input image is a small tensor, with usually square shape, of dimension (n, n, c) with $n \leq \min(w, h)$.
- The convolution operation is done as follows: the kernel is sliding over the bigger input tensor, at each location (x, y) of the centre of the kernel the element-wise multiplication between the entries of the kernel and the entries of the input 'covered' by the kernel is performed, and the results are all summed up. The final value is the (x, y) entry of the feature map. Usually the kernel size n is an odd number: in this way the (x, y) position of the centre of the kernel can be well-defined. This procedure, in the case of 2-D matrices, is represented in fig.2.7.

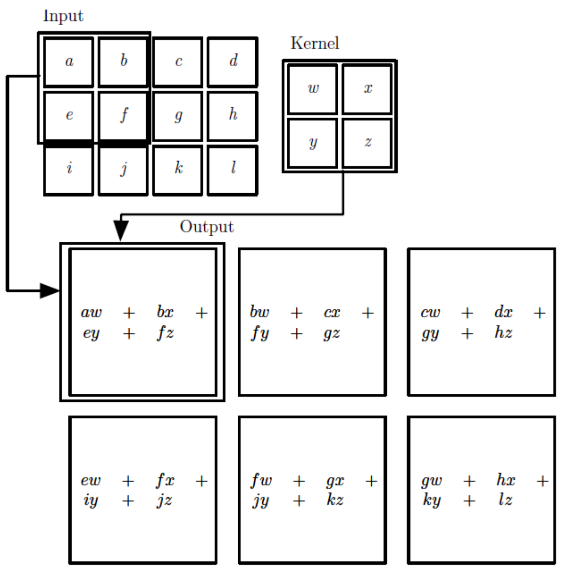


Figure 2.7. Example of a convolution. Adapted from [6].

The application of a convolution results in specific image processing functions, depending on the values of the entries of the kernel. For example, some kernels are well-defined for operations like blurring, edge detection and sharpening. One of the main advantages of a Deep Learning approach is that the values of the filters applied in the network are automatically learned during the training process and not defined by the user.
Another important aspect must be pointed out. When using FC networks with multidimensional inputs, each layer performs a classical matrix product, reflecting the fact that each neuron is connected to every one of the next layer. This results in an huge number of weights also for reasonable small input tensors. On the other hand, in CNNs the matrix multiplication is substituted by the convolution with a smaller matrix, the kernel, resulting in the fact that neurons in the current layer are connected to only a small region of the layer before. In this way the number of weights considerably decreases, resulting in only the values of the filters applied. For this reason, CNNs are the preferred solution for solving CV tasks.

2.5.2 Padding

As explained before, the centre of the kernel slides over the input tensor and entries in the neighbourhood of its (x, y) position are taken into account for the convolution. The problem arises when scanning the contour of the input matrix: in this case some entries of the kernel lies outside of the input tensor and the convolution can not be correctly defined. Padding is the solution for this problem. There are two main possible strategies:

- Replicate the values at the border of the input matrix in the output one. Doing that, the starting position (x, y) must be such that all the kernel entries lie inside the input matrix.
- Zero Padding. Add rows and columns of zeros outside the border of the matrix, in order to perform the convolution starting from the upper-left entry of the input tensor.

Using padding has the advantage of generating outputs with larger dimensions than without using it, due to the reduction in dimension implied by the convolution operation itself. This is a good outcome, in particular with very deep networks: in these cases consecutive convolutions may reduce too much rapidly the size of the input matrices of the deepest layers, resulting in too much loss of information. Padding avoid this issue.
To conclude this section about convolution, it is worth to mention three others parameters to set when building a convolutional layer:

- Depth. It is the kernel's third dimension, in the case of the first convolutional layer it is equal to the channel depth of the input image, for a generic hidden layer it is equal to the depth of the output of the previous layer.
- Number of filters. It is the number of kernels used in the convolutional layer. Due to the fact that each filter generates a 2-D map (remember that kernel and input tensor have equal depth), this number defines the depth of the output of the layer because the 2-dimensional feature maps are stacked one over the other along the third dimension, see fig. 2.8.
- Stride. It defines the step used when the kernel slides above the input matrix. Must be defined for both x and y directions, typical values are 1 and 2. Along with the padding, it affects the size of the output tensor.

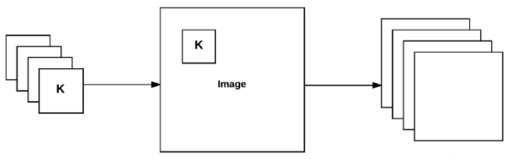


Figure 2.8. Convolution with multiple kernels. Adapted from [20].

2.5.3 Typical Layers in CNNs

Besides convolutional layers, other types of layer are almost always present in the structure of a CNN. They are:

- **Pooling layer.** As said before, the application of a convolutional layer with stride > 1 generates a reduction of the dimensions of the input volume. Another way is by using a pooling layer. As in the case of convolution, a kernel size and a stride have to be defined by the user, typical are values are $(2, 2)$ and 1 or 2, respectively. Sometimes, larger kernel size such as (3,3) are used in case of large input tensor. Inside of the kernel two types of operation can be performed: *max* or *average* operation. In the first case, the maximum of the entries inside the kernel is taken, in the second, the average of the entries is taken. It is important to notice that, recently, the trend is to discard the use of pooling layers [35] and just one average layer is used at the end of the network.

- **Activation layer.** After the convolution operation, an activation layer is used to apply the non-linear function, usually ReLU or one of its variants.

- **Batch Normalization layer.** Introduced by S. Ioffe and C. Szegedy [11], a batch normalization layer is used to normalize the input tensor before going to the next layer. The output tensor will have zero-centred values and unit variance. It us used to reduce the number of training epochs, leading to lower final loss. It helps also in reducing overfitting, because it makes easier the tuning of the hyperparameters.

Chapter 3

Hardware setup

The chapter presents a brief description of the target board for the hardware implementation of the face detector. More focus is given to the CNN accelerator core that performs convolution operations.

3.1 EVDK

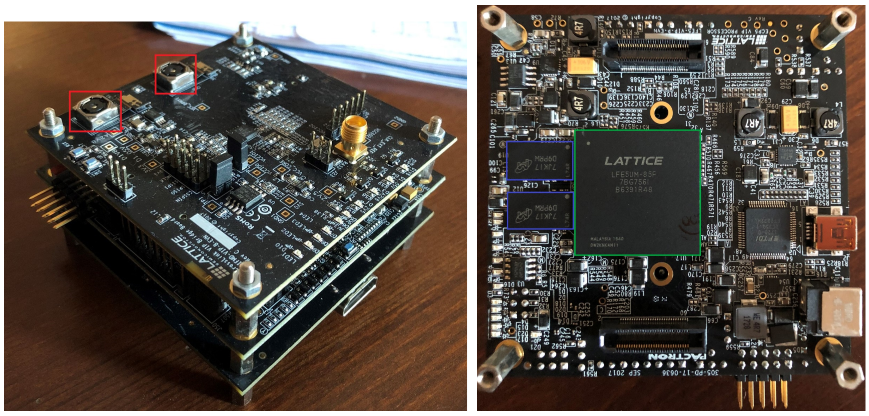
The Embedded Vision Development Kit (EVDK) is a modular platform for video processing at the Edge. It is composed by three different boards connected one over the other:

1. CrossLink VIP Input Bridge Board [23], composed by the the CrossLink FPGA that receives the data from the two on-board integrated camera sensors Sony IMX214 [34], bridging the CSI-2 interface of the cameras with a parallel one. Its main task is to simply send the pixel data, in the RAW10 format, to the next board for successive operations. The bitstream file for programming the FPGA can be downloaded from the Lattice Semiconductors™ website.

2. ECP5 VIP Processor Board [25], including an ECP5 family FPGA, that is the core for the analysis of images, dual DDR3 interface, SERDES interface and LVDS/MIPI Transmitter/Receiver interface. The FPGA is described in more details in next section 3.2. It also has two integrated Double Data Rate (DDR) memories for a total capacity of 2 Gb.

3. HDMI VIP Output Bridge Board [26], its key component is the HDMI Deep Color Transmitter, for transmitting video signals to an external monitor.

As it is possible to see, the EVDK contains everything for the data acquisition, processing and visualization, allowing for the development of CV prototypes for edge devices.



(a) EVDK setup. (b) ECP5 VIP Processor Board.

Figure 3.1. (a) The EVDK platform is shown, from the top to the bottom there are: the CrossLink VIP Input Bridge Board, ECP5 VIP Processor Board and HDMI VIP Output Bridge Board. In red are highlighted the 2 camera sensors. (b) The middle board is shown. Highlighted in green the ECP5UM-85 FPGA and in blue the two DDR3 DRAM.

3.2 ECP5 FPGA

The most important part of the all system is the ECP5UM-85 FPGA [24], the one that has to be programmed by the user in order to perform all the operations needed for the analysis of the input data. The characteristics of the device are reported in table 3.1.

LUTs (K)	84
sysMEM Blocks (18 Kb)	208
Embedded Memory (Kb)	3744
Distributed RAM Bits (Kb)	669
18 X 18 Multipliers	156
SERDES (Dual/Channels)	2/4
PLLs/DLLs	4/4

Table 3.1. Summary of the resources for ECP5UM-85 FPGA.

This FPGA allows the integration of systems without caring too much on the resource usage thanks to the almost 85 thousands of LUTs. In addition the sysDSP

can perform 54-bits ALU operations, advanced 18 x 36 MAC and 18 x 18 Multiply-Multiply-Accumulate (MMAC) operations essentials for the hardware implementation of convolutional layers of CNN. The core power supply is equal to 1.2 V, limiting the power consumptions.
The main limiting factor is the integrated memory equal to 3.744 Mb. This memory is divided into 208 blocks, each of them with a dimension of 18 Kb, used for the configuration of a large variety, in depth and width, RAM and ROM memories. Next figure 3.2 shows a simplified schematic of the structure of the FPGA and summaries the various features available.

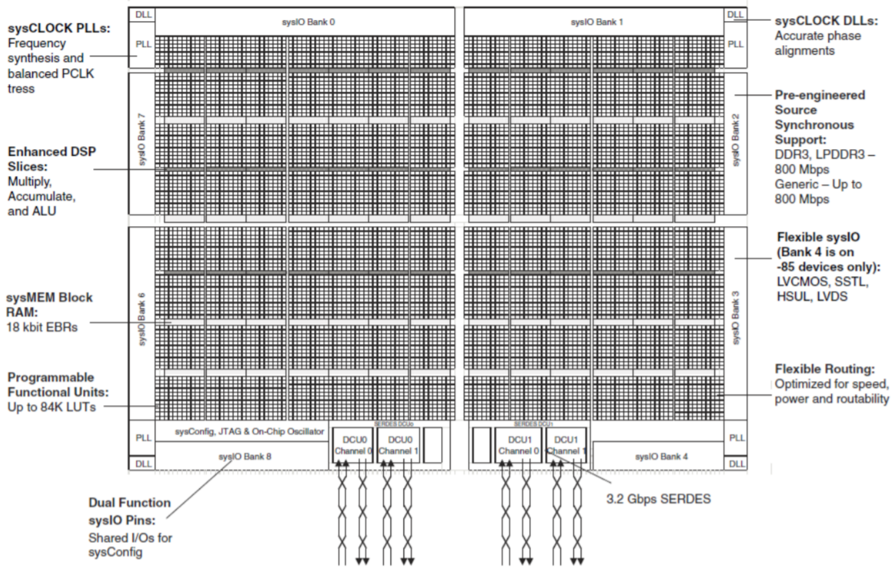

sysCLOCK PLLs: Frequency synthesis and balanced PCLK tress
Enhanced DSP Slices: Multiply, Accumulate, and ALU
sysMEM Block RAM: 18 kbit EBRs
Programmable Functional Units: Up to 84K LUTs
Dual Function sysIO Pins: Shared I/Os for sysConfig
sysCLOCK DLLs: Accurate phase alignments
Pre-engineered Source Synchronous Support: DDR3, LPDDR3 – 800 Mbps Generic – Up to 800 Mbps
Flexible sysIO (Bank 4 is on -85 devices only): LVCMOS, SSTL, HSUL, LVDS
Flexible Routing: Optimized for speed, power and routability
3.2 Gbps SERDES
sysIO Bank 0
sysIO Bank 1
sysIO Bank 2
sysIO Bank 3
sysIO Bank 4
sysIO Bank 6
sysIO Bank 7
sysIO Bank 8
sysConfig, JTAG & On-Chip Oscillator
DCU0 Channel 0
DCU0 Channel 1
DCU1 Channel 0
DCU1 Channel 1
DLL
PLL

Figure 3.2. Simplified schematic of the Lattice ECP5UM-85 FPGA. Adapted from [24].

3.2.1 CNN Accelerator Core

For the implementation of a face detector based on CNN a module capable to reproduce the convolution operation is necessary. In general, it is quite easy to map the convolution in hardware, it essentially is the subsequent application of MAC operations. In addition to its high computational capability, the ECP5 FPGA has been chosen because there is the possibility of directly use an Intellectual Property (IP) for a CNN Accelerator Core [22]. Some of its main features are:

- support for convolutional, max and average pooling, batch normalization and FC layers,
- configurable bit width of weights (16-bits, 1-bit),

- configurable bit width of activations (16/8-bits, 1-bit),
- configurable number of engines and memory blocks,
- optimization for 3x3 2D convolutions,
- configurable input byte mode (signed, unsigned, disable),
- supports MobileNet.

The figure 3.3 shows an high level block representation of its internal structure.

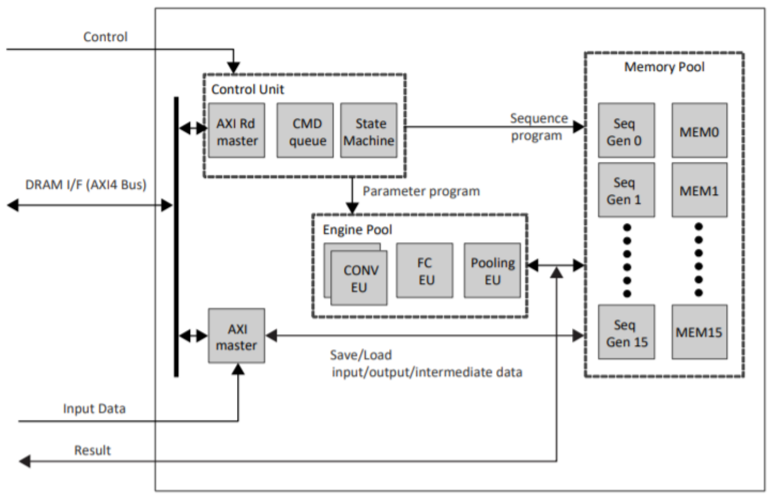


Figure 3.3. Functional block level representation of the CNN Accelerator Core IP. Adapted from [22].

Different sub-blocks are present:

- the control unit that organizes the operations for the other modules,
- the engine module where actual computations are performed. The number of engines can be configured by the user with a maximum of 8 engines,
- the memory pool, used for saving partial data during computation. Also their number can be configured by the user, the recommended number is equal to the double of the number of engines.

The next figure 3.4 shows the external interface ports of the CNN Accelerator module. It has clock and reset ports, a set of control ports including the i_start for starting the computation and the o_rd_rdy that is equal to 1 when the module is in the idle state and ready for starting a new computation, a Lattice Memory Mapped Interface (LMMI) [28] for writing input data, a DRAM interface working with an AXI4 protocol, the result ports and a status signal for checking in which state is the module.

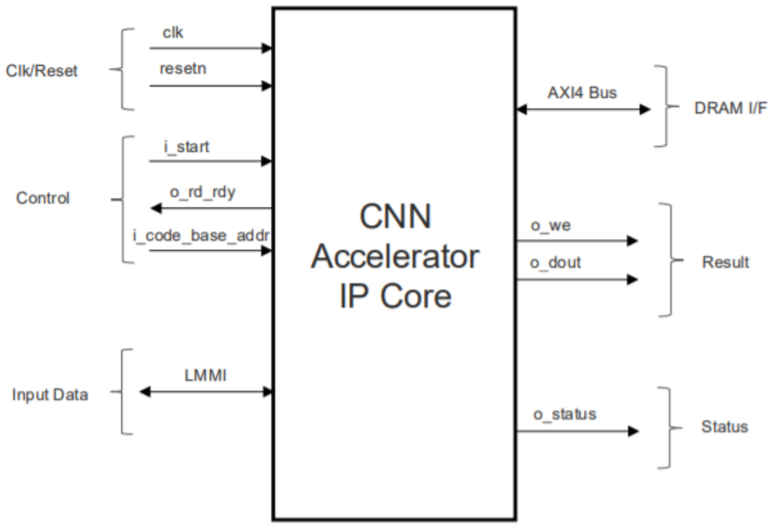


Figure 3.4. Interface ports for the CNN Accelerator Core IP. Adapted from [22].

The typical operational procedure to follow is:

1. Assertion of the reset.
2. De-assertion of the reset, i_start must be de-asserted too.
3. The command code for the operations to be performed has to be loaded on the DRAM memory integrated on the ECP5 VIP Processor Board. The command code is generated through the compilation of the network structure described on high level language, such as Python. i_code_base_addr specifies the starting address of the command code to be read.
4. Check of o_rd_rdy status. It must be 1, otherwise start again from step 1.
5. The input data is written into the internal memories of the module.
6. i_start is asserted and o_rd_rdy must be equal to 0 after the assertion.
7. De-assertion of i_start.
8. Checking of o_we, the data coming from o_dout has to be collected while o_we is equal to 1.
9. Repeat from step 5.

3.2.2 Tools Used

The two main tools used for the hardware implementation are the NN compiler and the Lattice Diamond software.

NN Compiler

The Lattice NN compiler [29] is used to generate the command code for the the CNN Accelerator module by taking structure and weights of the model defined in Caffè, Tensorflow or Keras. In addition to this, it allows the user to define the width of the input data and how many bits are used for the fractional part of the fixed point representation. Also, the user has to define the number of engines and memories used for the computation, these two settings must be the same when instantiating the IP on the Hardware Description Language (HDL) code. After the analysis and compilation of the network, information regarding the number of clock cycles needed for the computation are given, in addition to a report about how the input data is saved in the internal memories. The format of the command is a sequence of 32-bits data with additional optional parameters, as shown in fig. 3.5.

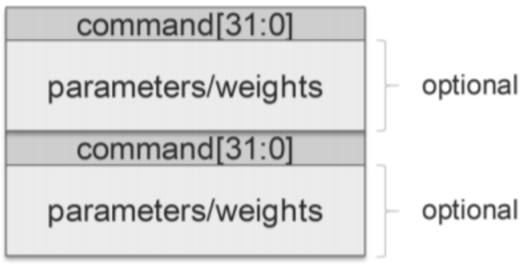


Figure 3.5. Command code format for the CNN Accelerator Core IP. Adapted from [22].

The NN compiler permits also to simulate the model if an input sample is provided. Thanks to this feature, it is possible to directly compare the output of the network in the software implementation, with a floating point representation of numbers, with the corresponding fixed point one, and to measure how much the results differ.
Finally, a debug option is available for checking directly on the device the data written and read from the internal registers. For this feature is necessary to substitute the HDMI VIP Output Bridge Board of the EVDK with the USB3-GbE VIP IO Board [31] and to connect it to an external computer for programming it. Unfortunately, after several attempts, the result was that this feature was not working, probably due to a conflict between some drivers. It was confirmed by the

previous experience of people in the laboratory and the seller company, i.e. Lattice Semiconductors™, does not provide any solution when contacted. This forbids the checking of the correctness of results from the CNN core.

Lattice Diamond Software

The Lattice Diamond software is used for the synthesis, place and route and generation of the .bit file to be loaded on the board, from the HDL code of the system. In addition to these features, it allows the generation of the IP modules from an large set available, including:

- audio, video and image processing modules such as byte to pixel converters,
- communications modules like Ethernet Media Access Controllers,
- connectivity modules,
- Digital Signal Processing (DSP) modules, including the CNN Accelerator Core and,
- controllers for external peripherals like the DDR3 controller for the integrated DRAM on the ECP5 board.

Chapter 4

Software Implementation

The chapter presents a short introduction about face detection problem. Then, the chosen network is analysed and details about the training are explained. In the second part, modifications applied to the reference design are discussed and the results are presented.

4.1 CNN for Face Detection

As stated before, currently CNNs constitute the state of the art for face detection algorithms, thanks to their learning capability. In order to obtain a strong and generalized model, a large training dataset with images of different types is needed. The challenges connected to face detection task can be related to the following attributes:

- **Scale**. The most intuitive problem when detecting faces in images is that they can appear at different scales. In general, small faces are more difficult to be detected.
- **Pose**. It describes the relative position between the face and the camera (frontal, 45 degrees, profile, etc.). In some cases, a face can be partially occulted.
- **Structural components**. It includes beards, glasses, mustaches that can be present or not in a face. In addition they can vary in colour and shape, and can hide some facial details.
- **Occlusion**. Other objects can partially mask a face, or in case of a group of people, a face can hide another one.
- **Facial expression**. Depending on it the face appearance can change.

- **Imaging conditions**. It includes lighting and characteristics of the camera sensor.

An optimal face detector should be able to correctly detect faces with all these different conditions. This result can be achieved if also the training dataset includes all the various factors. For the project, the reference dataset used is the WIDER-FACE dataset, that has been demonstrated to be extremely challenging with respect to other datasets [37]. The dataset consists on 32203 images with 393703 labelled faces with variations in pose, scale and occlusion. The images are divided in 61 different classes, and for each class, 40%, 10%, 50% of images are used for training, validation and testing, respectively.
Several frameworks have been evaluated by using this dataset, from the seminal work of Viola-Jones [36] to more complicated recent detectors, as reported in [39] and [41], capable to reach the best accuracy results. Among the over 30 methods reported, the chosen one is the Multitask Convolutional Neural Network.

4.2 Multitask Cascaded Neural Network

When dealing with EdgeAI applications, the focus is much more on the portability of the detector than on its accuracy. For this reason, high-accurate models with complex structure and large number of weights are not suitable for sensing application. The choice of the Multitask Cascaded Neural Network (MTCNN) [40] follows this reasoning: it performs quite well among all the models and it has a simple structure that can be loaded on small-memory devices.

Typical object detection, and so also face detection, networks have a structure based on a backbone for the feature extraction and only the last layers are modified depending on the needs. Usually, this kind of models have a very high detection accuracy but also a number of parameters that make them unsuitable for portable applications. The most common examples are the VGG-16 [33] and its evolution VGG-19, MobileNet [10] and MobileNetV2 [21] models. The next table 4.1 reports the number of parameters for these models and compares them with the MTCNN.

	VGG16	VGG19	MobileNet	MobileNetV2	MTCNN
Parameters	138357544	143667240	4253864	3538984	492410

Table 4.1. Comparison on number of weights for different models.

From the table it is possible to see that the MTCNN has a very low number of parameters compared to the other methods, making it suitable for a portable face detector on a small-sized memory device, such as an FPGA. In parallel it obtains reasonable results in terms of detection accuracy over the WIDER-FACE validation dataset, as reported in the next table.

	Easy dataset	Medium dataset	Hard dataset
MTCNN accuracy	84.8%	82.5%	59.8%

Table 4.2. Summary of the detection accuracy of the MTCNN.

4.2.1 Network Structure

The structure of the MTCNN follows the previous works [16] and [38], where a cascade of networks is used to locate and refine the position of faces in images. By paying attention to the intrinsic correlation between the problem of face detection and face alignment, a multi-task network was designed addressing both problems and, in this way, reaching better results with respect to the previous solutions of the time.
The detector consists of three stages:

1. Proposal Network (P-Net).
2. Refine Network (R-Net).
3. Output Network (O-Net).

P-Net

The aim of the first stage is to generate region proposals for the successive steps. The functionality of the P-Net can be explained as follows: given an input image of any size, a sliding window of dimension 12x12 pixels moves over the image with stride equal to 2 and, for each region of 144 pixels, three different outputs are generated. They are:

- Face classification output with a depth of two: the probability of not-being a face and being a face for that region.
- An output of depth equal to four. They are the bounding box regression outputs, essentially the corrections for the 4 coordinates of the 12x12 region in order to correctly surround a face. Each proposal region is identified by the top left and bottom right coordinates (x_1, y_1) and (x_2, y_2), respectively.

- The facial landmark localization output with a depth of ten. As for the bounding box regression, they are the corrections for the coordinates of the 5 facial points for face alignment, They are the two eyes, the nose and the two corners of the mouth.

All the process of the sliding window over the input image is actually equal to apply a CNN to the input image, generating the three kinds of results. The structure of the network is reported in the following fig. 4.1.

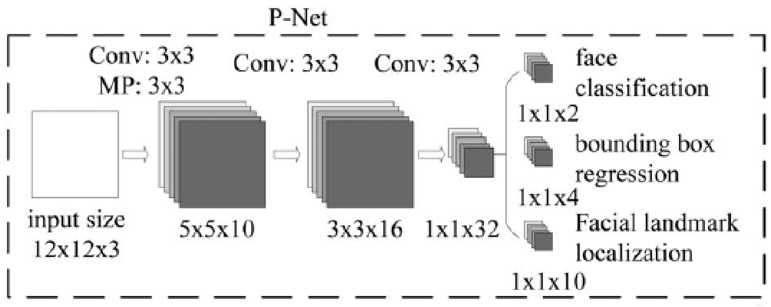


Figure 4.1. Structure of P-Net. Adapted from [40].

In the previous figure the input image is supposed to be of size 12x12x3, in this case there is obviously just one region to analyse and the output predictions are just vectors. In the case of generic size for the input image, the outputs will be tensors with 3 dimensions, where each point in the result matrices is the prediction for one region of size 12x12 in the input image. In this way, a generic point (x, y) in each output tensor refers to a specific region of the input image, and at the same position (x, y) in all the three outputs there are information about the classification, box regression and face localization for this specific window.

The structure of the P-Net is very simple, it consists of just 3 convolutional layers with kernel size equal to (3,3) and stride equal to 1; the first convolution is followed by a max-pooling layer with pool size of (3,3) and stride 2. The number of filters used in each convolutional layer is equal to the depth of tensors following its application (as explained in sec. 2.5.2). In this case 10, 16, 32 filters are used in the corresponding convolutional layers, and 2, 4, 10 filters for the three different outputs. The activation function after every convolution is the PReLU, except for the last layer. A network with this kind of structure is called *fully convolutional*, because the typical FC layer at the end is not used. This has been done in order to permit generic size for the input image: the dimension of the output tensors will vary depending on the input size image, while using a FC layer at the end would have required the definition of the number of output nodes when building the network. The goal of the P-Net is to generate, in a fast way, the candidate facial windows to be fed in the next stages, without being too much precise. This is why it has its *shallow* structure.

The input image of the whole face detector is firstly resized to different scales, generating an input image pyramid. The list of scales is defined by using the minimum face size that the user wants to detect and with a multiplication factor of 0.79. Scaled images are recursively generated until the condition

$$\min(height, width) \geq 12$$

is satisfied. Each single rescaled image has to be fed in input to the P-Net, in order to detect faces with different dimensions. It is important to notice that the network does not provide directly the coordinates of the proposal windows but just the correction to apply. So the boxes' coordinates in the original image size must be calculated from the coordinates of the points inside the output matrices for each scaled image (remember that each point represent a 12x12 window). This can be achieved by applying the following equations:

$$X_1 = (x \cdot S)/scale \qquad Y_1 = (y \cdot S)/scale \tag{4.1}$$

$$X_2 = (x \cdot S + cellsize)/scale \qquad Y_2 = (y \cdot S + cellsize)/scale \tag{4.2}$$

where:

- (X_1, Y_1) and (X_2, Y_2) are the top-left and bottom-right corner coordinates of the bounding box in the original image size,
- (x, y) are the coordinates of one point in the output matrix linked to a specific scaled image,
- S is the stride, in this case is equal to 2,
- *cellsize* is the dimension of the sliding window, in this case equal to 12,
- *scale* is the rescaling factor applied to the original input image.

Obviously, not all of the generated bounding boxes will contain a face, for this reason a threshold is defined by user: only the regions with a probability of being a face higher than the threshold are considered for next computation.
In general a CNN does not produce probabilities at its end, these are the results of the application of the *softmax* function to the face classification output.
It is a function $\sigma : \mathbb{R}^K \longrightarrow \mathbb{R}^K$ defined as:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \tag{4.3}$$

By using words, given a vector, the application of the softmax returns a vector of the same length with values normalized in the range [0,1] that can be interpreted as probabilities by humans. By doing that, also the threshold has to be a value between 0 and 1. This value has to be decided by the user, with value of 0.5 all the regions that are classified as faces by the P-Net will be taken, higher values like 0.6 and 0.7 allow to a first reduction of the proposals for next steps.

As said before, each scaled image is fed inside the P-Net, a common result is that multiple bounding boxes, close one to each other, surround the same face. It has not much sense to consider all these boxes for successive analysis, so a filtering process is used, called Non Maximum Suppresion (NMS).
The NMS algorithm is based on the Intersection over Union (IoU) ratio, defined as the ratio between the intersection area and the union of the areas of two boxes. This number gives a value of how much two regions overlap one with the other. The NMS algorithm can be summarized as:

1. Given the proposals with face probability above the threshold, sort them in descending order of face probability.
2. Take the proposal with the highest face probability, and calculate the IoU between this box and all the successive ones.
3. All the boxes for which the IoU value is above a certain threshold, usually 0.5 or 0.6, are discarded.
4. Move to the successive box with the highest face probability and repeat the previous steps until the last remaining box.

The NMS is applied to the results of each scaled image and also after that all the proposals are gather together. The main goal is the further reduction of the number of proposals to decrease the calculation time of the next two stages. After the generation and the filtering of the bounding boxes, their coordinates are adjusted by using the bounding box regression output of the network thanks the following equations

$$X_1' = X_1 + \Delta X_1 \cdot W \qquad Y_1' = Y_1 + \Delta Y_1 \cdot H \tag{4.4}$$
$$X_2' = X_2 + \Delta X_2 \cdot W \qquad Y_2' = Y_2 + \Delta Y_2 \cdot H \tag{4.5}$$

where:

- (X_1', Y_1') and (X_2', Y_2') are the corrected coordinates for the bounding boxes,
- (X_1, Y_1) and (X_2, Y_2) are the initial coordinates for the proposals, generated by using the previous eq. 4.1 and 4.2,

- ΔX_1 ΔY_1 ΔX_2 ΔY_2 constitute the bounding box regression output of the network for a specific 12x12 window,
- W and H are width and height of the box, calculated as $X_2 - X_1$ and $Y_2 - Y_1$ respectively.

Finally, all the steps done in the first stage can be summarized as:

1. Generate the image pyramid for the input image.
2. Each scaled image is presented to the input of the P-Net.
3. By using the face classification output, the regions with face probability above the threshold are kept, for each scaled image in input, and their bounding box coordinates are generated by using eq. 4.1 and 4.2.
4. NMS is applied first to proposals generated from the same scaled image and then to the regions from all scaled images gather together.
5. Bounding box calibration is performed through eq. 4.4 and 4.5.

R-Net

After the first stage, the detector has a set of proposals defined by four coordinates and for each one the associated face probability. The second stage takes each proposal, resizes it to a 24x24 image and feeds it inside the R-Net. The structure of the network is reported in the next fig. 4.2.

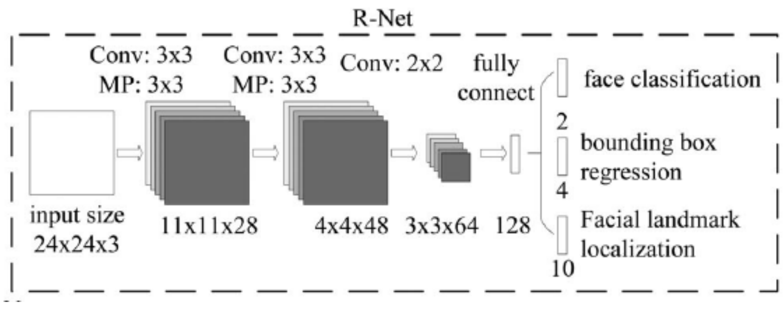


Figure 4.2. Structure of R-Net. Adapted from [40].

As already said, it takes an input image of size 24x24 and, in sequence, the following layers are applied:

- Two pairs of convolutional layer and pooling layer. The two convolutions have a kernel size of (3,3), stride = 1, 128 and 48 filters respectively. The max-pooling layers have pool-size = (3,3) and stride equal to 2.

- A convolutional layer with 64 filters, kernel size equal to (2,2) and stride equal to 1.
- A FC layer with 128 neurons. From here, the three outputs branch out, each one with another FC layer whose number of nodes is 2, 4 or 10 depending on the output type.

The activation function of convolutional layers is the PReLU, while for the FC layers no activation function is applied except for the face classification output where the softmax is used. The first difference from the P-Net is that this is not a fully convolutional network, there is no need of doing that because the input size is fixed in this situation. Also, the number of filters applied is higher from before, because the network has to make better predictions about its input image content, so a finer extraction of information from the input has to be done. The main task of the R-Net is to discard a large number of false candidates coming from the P-Net proposals.

For each input image, a decision is made depending on the face probability. Also in this case, if it is higher than a defined threshold, it is taken for successive steps. The bounding box calibration is performed by applying the corrections coming from the bounding box regression output of the R-Net to the coordinates of the window from the previous stage. It is done in the same manner as in eq. 4.4 and 4.5. After all the P-Net proposals have been analysed, the remaining boxes are submitted to a NMS process to further reduce the number of results. At the end of R-Net stage, the corrected coordinates of remaining proposals are saved.

O-Net

The last stage of MTCNN is similar to the previous one. It takes the proposals from the R-Net, resizes them to images with 48x48 size, and feeds them at the input of the O-Net. The structure of the network is shown in fig. 4.3.

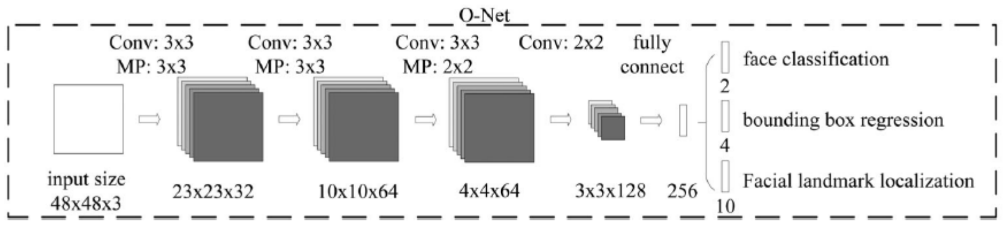


Figure 4.3. Structure of O-Net. Adapted from [40].

This network has to be the most precise of the three, because is the last one which generates the final results. This is reflected in its structure, that is the one with more layers and weights. In particular:

- Three pairs of convolutional and pooling layers. Convolutions have kernel size (3,3), stride 1 and 32, 64, 64 filters respectively. After each convolution, max-pooling is applied with pool size (3,3) for the first two pairs and (2,2) for the last one. The stride is equal to 2 for all of them.
- A last convolutional layer with 128 filters, kernel size (2,2) and stride 1.
- A FC layer with 256 neurons. From here the three outputs are generated by using FC layers with 2, 4 and 10 output nodes.

The network is similar to the R-Net, and same considerations can be applied regarding the activation functions. Also in this case, the main task of the network is to eliminate last possible false positive proposals and to do bounding box regression, both of them with more accuracy. As probably noticed, in the previous paragraphs about P-Net and R-Net the facial landmark output is never used for performing successive calculations. In fact, although it is present, the regression is never performed in early stages but just in the last one.

The process is the same as in R-Net case. Firstly a decision is made on the input resized proposal depending on the face probability output. If it is higher than the last threshold, the proposal is saved and bounding box calibration is performed using eq. 4.4 and 4.5. The remaining proposals are subjected to NMS to discard highly overlapping boxes.

Finally, the next figure 4.4 summaries all the pipeline of the MTCNN based face detector, with a visual example.

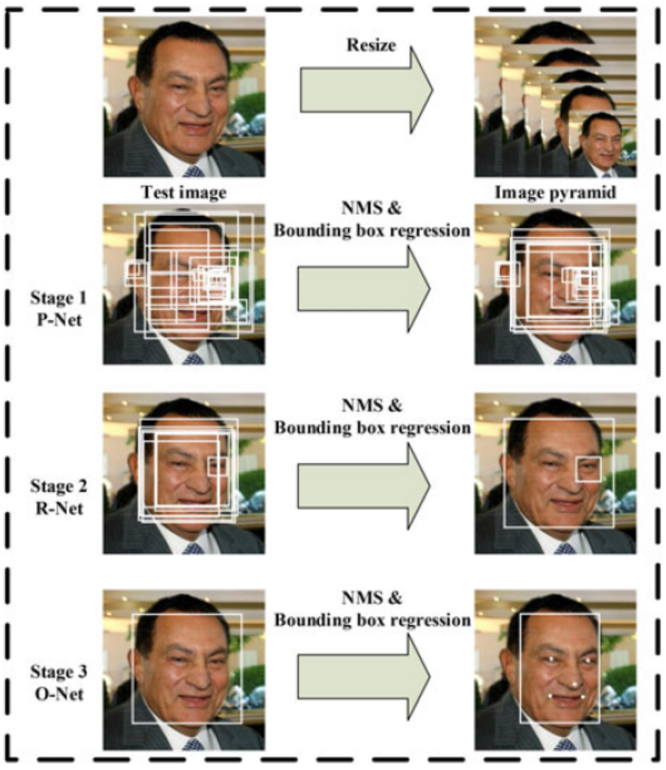


Figure 4.4. Pipeline of the cascade structure. Adapted from [40].

4.2.2 Training Process

As explained before, the MTCNN detector has three different outputs, and for each of them, a loss function has to be defined for the training process. Just to remind, the loss function is a metric that measures how much the predictions differs from correct values, see sec. 2.4.1 for more details. In the reference work done in [40], the training dataset for face classification and bounding box regression was prepared from the WIDER-FACE training dataset, while for facial alignment the CelebA dataset [18] was used.

- For the P-Net, from WIDER-FACE images, random crops of dimension 12x12 are taken and used for the face classification and bounding box regression. For the face alignment, faces are cropped from CelebA dataset. In particular there are four types of samples:

  Negatives for which the IoU with any ground-truth face is lower than 0.3.

  Positives for which the IoU is higher than 0.65 to a ground-truth face.

  Partial with IoU in the range $[0.4, 0.65]$ to a ground-truth face.

  Landmark faces samples with the five facial landmark labels.

- For the R-Net, the first stage is applied to WIDER-FACE images to generate negative, positive and partial samples depending on the previous values of IoU. So in this case, all the results from the P-Net are taken and labelled depending on the IoU. For the facial landmark, the P-Net is applied to CalebA images.
- For the O-Net, the process is the same as before. The first two stages are used to generate the different types of samples for the training.

The three loss functions used for training are:

1. $$L_i^{det} = -(y_i^{det}\, log(p_i) + (1 - y_i^{det})\, log(1 - p_i)) \tag{4.6}$$

   For the face classification task the loss is formulated as a binary cross-entropy loss, where p_i is the face probability given by the network for the i^{th} sample and $y_i^{det} \in \{0, 1\}$ is the ground-truth label for the same i^{th} sample.

2. $$L_i^{box} = \parallel \hat{y}_i^{box} - y_i^{box} \parallel_2^2 \tag{4.7}$$

   For the bounding box regression problem, the network predicts the offset between the proposal candidate coordinates and its nearest ground-truth window. The loss function is formulated as the Euclidean distance, where $\hat{y}_i^{box} \in \mathbb{R}^4$ are the results given by the network while $y_i^{box} \in \mathbb{R}^4$ are the ground-truth coordinates, both referring to the i^{th} sample.

3. $$L_i^{landmark} = \parallel \hat{y}_i^{landmark} - y_i^{landmark} \parallel_2^2 \tag{4.8}$$

   For the facial landmark task, the problem is again a regression one, with the Euclidean loss between the network results $\hat{y}_i^{landmark}$ and the ground-truth coordinates $y_i^{landmark}$ for the i^{th} sample. There are five facial points, each one defined by 2 coordinates so $\hat{y}_i^{landmark} \in \mathbb{R}^{10}$ and $y_i^{landmark} \in \mathbb{R}^{10}$.

This kind of process, where a network has multiple outputs with different tasks, is called *multi-task* training. In these type of training processes the total loss is not just the sum of single losses, because it depends also on the type of the sample fed inside the network. For example, in the case of a negative sample, only the detection loss 4.6 makes sense to be considered, in case of positive sample the bounding box regression loss 4.7 has to be added, while in the case of partial sample only the box regression loss is used. The overall learning loss can be formulated as:

$$\sum_{i=1}^{N} \sum_{j \in \{det,\ box,\ landmark\}} \alpha_j \beta_i^j L_i^j \tag{4.9}$$

where:

- $\beta_i^j \in \{0,1\}$ is a sample type indicator. Practically it makes sure that correct losses are considered depending on the sample type;
- α_j is the task importance. For P-Net and R-Net

  $$\alpha_{det} = 1 \qquad \alpha_{box} = \alpha_{landmark} = 0.5$$

  while for O-Net

  $$\alpha_{det} = \alpha_{landmark} = 1 \qquad \alpha_{box} = 0.5$$

- L_i^j is one of the task losses for the i^{th} sample;
- N is the number of training samples.

The optimization algorithm used is the SGD.

An important peculiarity of the work done in [40] is the implementation of *online hard sample mining*. This process consist in taking only the top 70% samples with highest loss for the computation of the gradient for the successive update of the weights. It is called in this way because it is embedded in the training process and only the samples that generates highest mis-detection are considered, so the most difficult, i.e. hard, samples. The procedure leads to better performance as reported in [32].

4.3 Training Implementation

The aim of this section is to describe how actually the training of the network has been done for the project, which simplifications have been made, and why.
The first thing that has to be pointed out is the fact that the authors of [40] did not provide any source code for the training while a Matlab implementation of the detector for the inference is available as open source code[1]. For this reason, after a first attempt of writing the training code from the beginning, due to the low quality results obtained and the difficulty in finding where exactly was the problem, the decision taken has been to use an already existing training setup available as open source code. The choice fell on a GitHub project[2], that has the best evaluation from users and declares to obtain results comparable with the ones reported in [40]. The repository includes codes for training data preparation, training and evaluation. The inference model is just the translation in Python, with the TensorFlow framework, of the Matlab code provided by the authors. The training code is implemented with also the online hard sample mining.

[1]`https://github.com/kpzhang93/MTCNN_face_detection_alignment`

[2]`https://github.com/AITTSMD/MTCNN-Tensorflow`

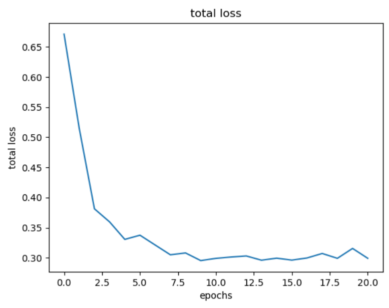
4.3.1 First Simplification

The original structure of the network expects to have three outputs. Considering the field of application for the project, i.e. EdgeAI applications, the facial landmark localization output is not essential, because the goal is to have a small-sized network instead of a precise one. Moreover, the elimination of this output reduces the operations to be done during the post-processing and allows a cross-checking of the actual gain in accuracy when it is present.

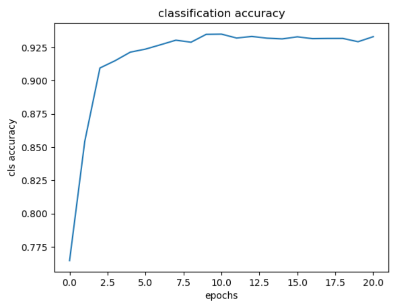
The elimination of the face alignment output is reflected in only the elimination of its related loss from the computation, the parameters α_{det} and α_{box} are left as before. The steps done can be summarized as following:

1. The training samples for the P-Net are prepared as explained in sec. 4.2.2. At the end have been generated:
    - 1000405 negative,
    - 457340 positive, and
    - 1129405 partial samples.
2. Training of the P-Net is performed for 20 epochs with batch size of 384.
3. The trained P-Net is used to generate the training dataset for the R-Net, resulting in
    - 771571 negative,
    - 123544 positive and,
    - 640518 partial samples.
4. Training of the R-Net for 26 epochs with batch size of 384.
5. The trained P-Net and R-Net are used to generate the dataset for the O-Net with:
    - 688001 negative,
    - 138143 positive and,
    - 218044 partial samples.
6. Training of the O-Net for 20 epochs with batch size of 384.

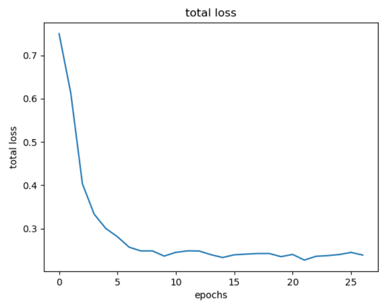
Each batch has a ratio between negative, positive and partial samples equal to $3 : 1 : 1$, in order to train the networks in conditions that emulate, as best as possible, the conditions with real images where the face regions are less than background ones. The optimization algorithm used is the SGD with momentum ($\alpha = 0.9$), the learning rate is equal to 0.001 and it decreases of a factor of 10 after 6, 14 and 20 epochs. Training is also implemented with an L2 regularization term for the weights in the convolutional layers, with $\lambda = 0.0005$, to improve the generalization. Also, model's weights are saved every two epochs, allowing the use of them in case of overfit of the final trained parameters. The next figure 4.5 reports the training total loss and detection accuracy for the three CNNs.



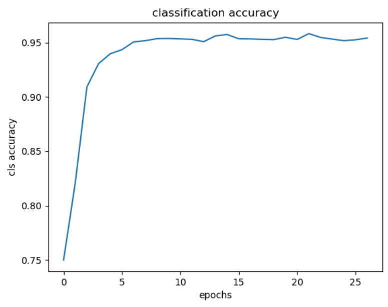
(a) Total training loss P-Net



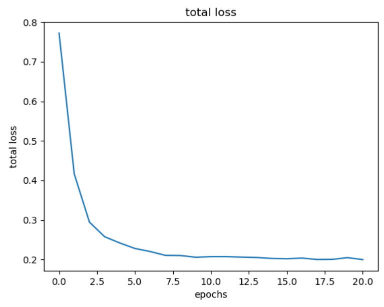
(b) Training detection accuracy P-Net



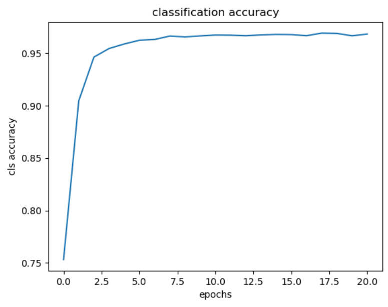
(c) Total training loss R-Net



(d) Training detection accuracy R-Net



(e) Total training loss O-Net



(f) Training detection accuracy O-Net

Figure 4.5. Training plots without Facial Landmark output.

As it is possible to see, the classification accuracy increases from about 92% of the P-Net to more than 95% in the case of O-Net.

In order to have a more effective result that can evaluate the efficiency of the detector, it has to be tested on new images. For this scope, it has been decided to use the WIDER-FACE validation dataset with 3226 images, because from the official website[3] it is possible to download a Matlab code allowing the users to generate the Precision-Recall (PR) curves for their model and to compare their results with existing benchmarks.
Precision and recall are defined as:

$$Precision = \frac{TP}{TP+FP} \qquad\qquad Recall = \frac{TP}{TP+FN} \tag{4.10}$$

where:

TP is the number of true positives, so the number of faces that are classified as such;

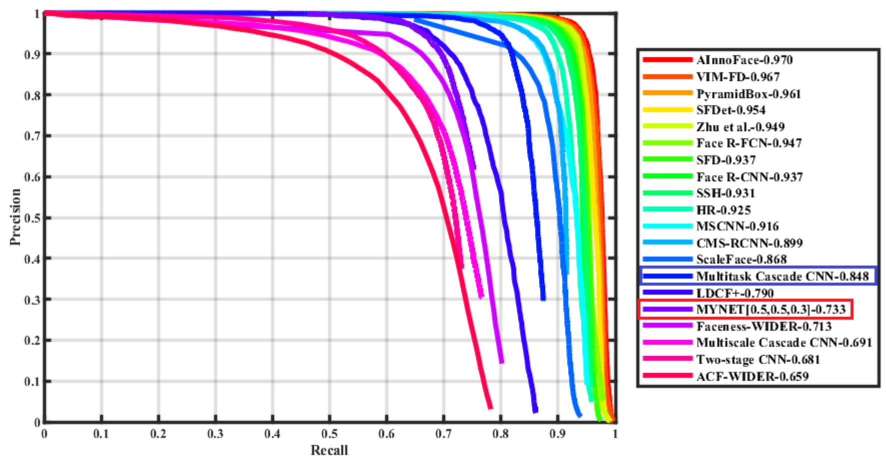
FP is the number of false positives, when the model predicts a face where there is not;

FN is the number of false negatives, that counts the times when a no-face is predicted but actually there is one.

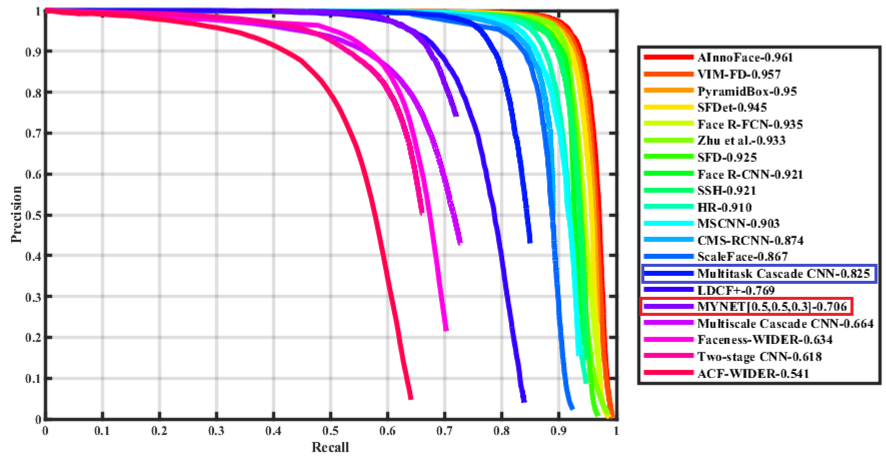
PR curves plot the values of precision and recall for different cut-off values, where the latter is defined as the threshold that discriminates when a result as to be interpreted as a face or not. Usually PR curves are not easy to understand if a model is working correctly or not, so also the mean accuracy over the samples is calculated, that is approximately equal to the area below the PR curve.

Next figure 4.6 reports the PR curves for the different partitions of the validation dataset where the easy one denotes the simplest images and the hard one the most difficult images. The face probability thresholds after each stage are [0.5 0.5 0.3] and the minimum face size is set to be 20x20 pixels. These setting are the same used by the authors when testing their model [40] on WIDER-FACE dataset. Furthermore, the three network models used are the ones after 20, 14 and 16 epochs of training, because, in the case of R-Net and O-Net, models trained for more epochs led to lower performances, probably due to an overfitting on training data.

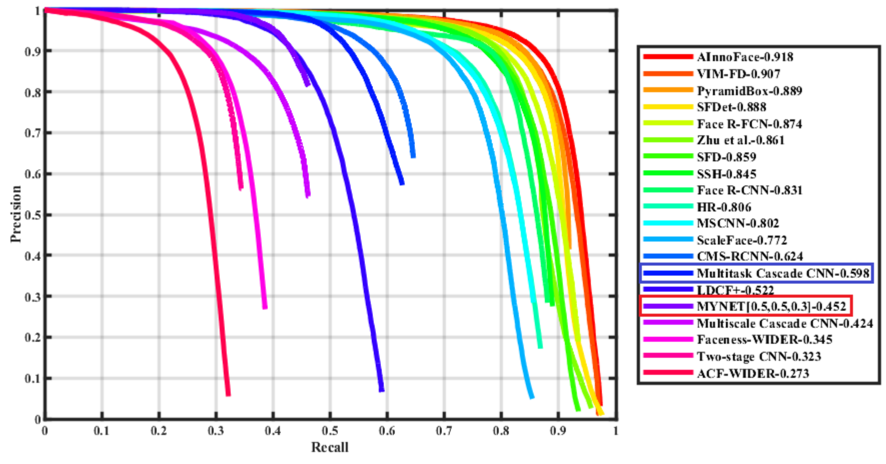
[3]`http://shuoyang1213.me/WIDERFACE/WiderFace_Results.html`



(a) Easy Dataset



(b) Medium Dataset



(c) Hard Dataset

Figure 4.6. PR curves for the model without Facial Landmark output.

	Easy dataset	Medium dataset	Hard dataset
MTCNN	84.8%	82.5%	59.8%
MTCNN modified	73.3%	70.6%	45.2%

Table 4.3. Comparison on the mean accuracy between MTCNN model and modified one.

As expected the modified model has lower performances on all the datasets compared to the reference model, confirming the importance of having also the facial landmark localization output implemented. Anyway, the results are good also in this case, with a loss in accuracy that ranges between 11,5% and 14,6% depending on the difficulty of images (see table 4.3), showing that this is a good point for successive simplifications for the hardware implementation.

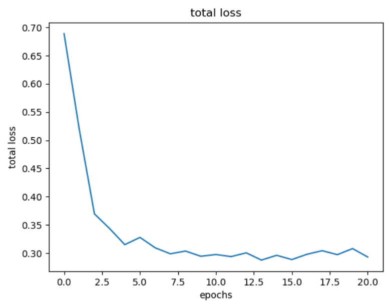
4.3.2 Second Simplifications

The second simplification made to the model has been forced due to tools restrictions. In particular, as explained in chapter 3, the network models must be compiled by the NN compiler in order to generate the set of instructions to be loaded on the FPGA for the CNN computation. Unfortunately, the software does not support PReLU and softmax activation functions, and networks with multiples outputs. The solution adopted is to use the ReLU activation after the convolutions, the softmax has been removed and applied outside the network for the successive calculations. For the generation of the output the structure has been modified: for the P-Net instead of two convolutional layers with 2 and 4 filters, one has been used with depth equal to 6, for the R-Net and O-Net the two FC layers have been substituted by only one with 6 neurons. These changes are reflected in the fact that correct portions from the output matrix and vectors have to be selected for the calculation of different losses.

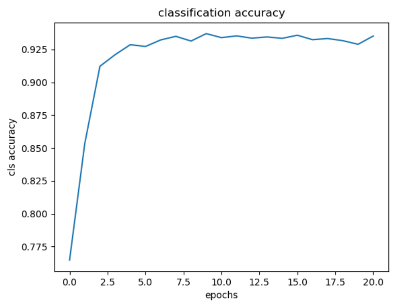
For the training, samples generated in sec. 4.3.1 are used and the the actual training was conducted for the same number of epochs, i.e. 20, 26 and 20, and with the same batch size of 384. The next table 4.4 resumes the training parameters, and results are plotted in fig. 4.7.

alpha	λ	starting *lr*	*lr* decay factor
0.9	0.0005	0.001	0.1

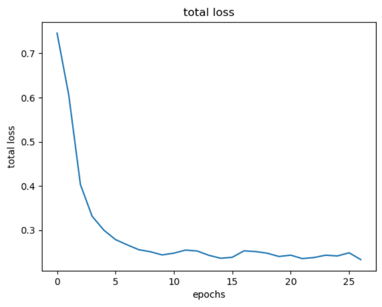
Table 4.4. Summary of hyperparameters for training process.



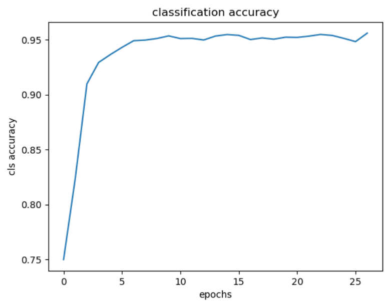
(a) Total training loss P-Net



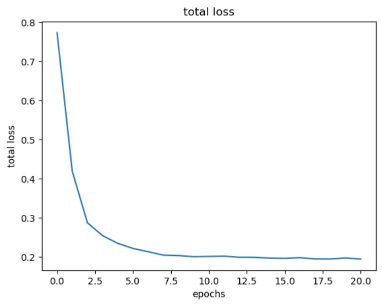
(b) Training detection accuracy P-Net



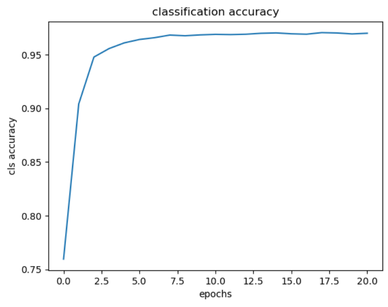
(c) Total training loss R-Net



(d) Training detection accuracy R-Net



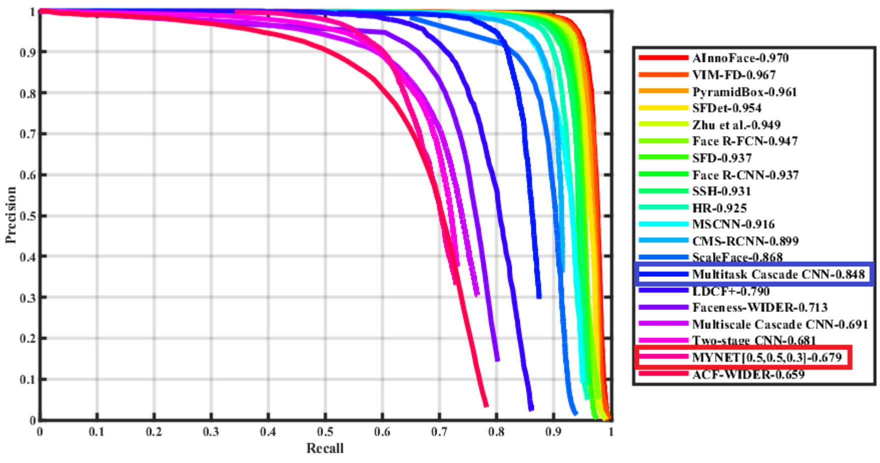
(e) Total training loss O-Net



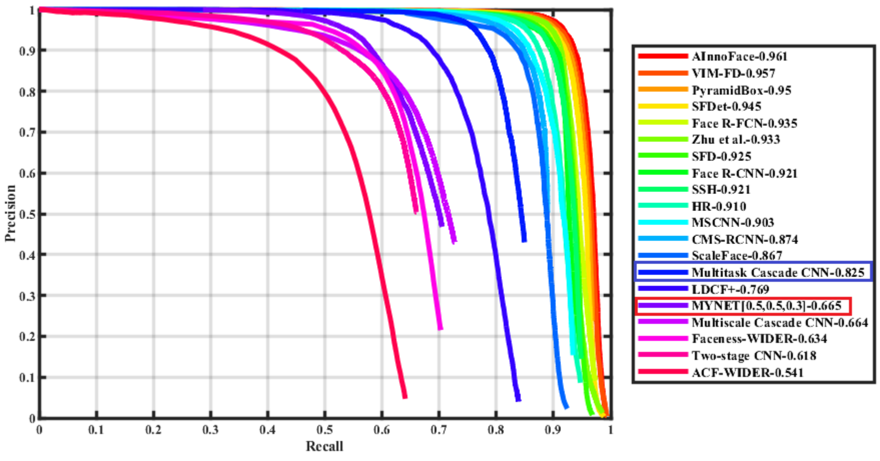
(f) Training detection accuracy O-Net

Figure 4.7. Training plots with previous simplifications applied.

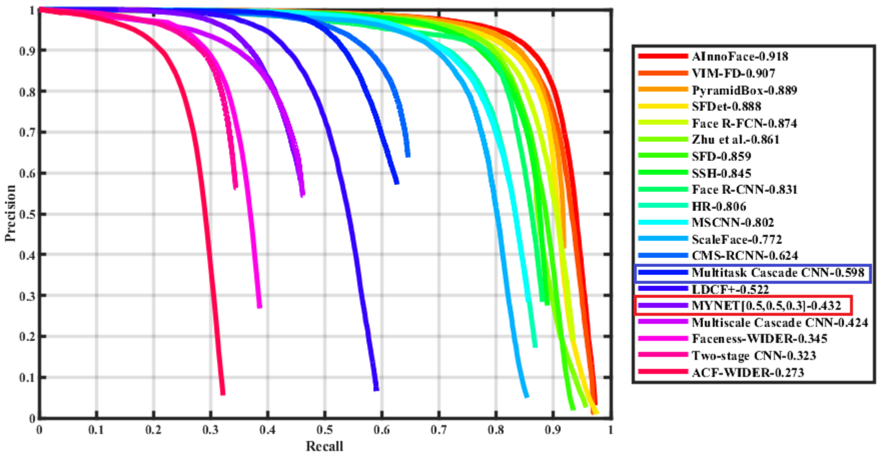
As before, these plots are not so explicative in showing how good the model actually is. Again, the detector has been tested on the WIDER-FACE validation dataset, in the same conditions as in 4.3.1. The PR curves are reported in the next fig. 4.8 and the comparison for the accuracy in table 4.5.



(a) Easy Dataset



(b) Medium Dataset



(c) Hard Dataset

Figure 4.8. PR curves for the MTCNN with second simplifications applied.

	Easy dataset	Medium dataset	Hard dataset
MTCNN	84.8%	82.5%	59.8%
MTCNN (1st simplification)	73.3%	70.6%	45.2%
MTCNN (2nd simplifications)	67.9%	66.5%	43.2%

Table 4.5. Comparison on the mean accuracy between MTCNN model and modified ones.

These further changes have generated another small decrease in accuracy, that varies between 5.4% and 2%. This was expected because now some training parameters, introduced by the PReLU activation function, have been removed. This result also confirms the fact that the usage of PReLU instead of ReLU improves the detection accuracy of models [9]. The models coming from this training setup are the ones used for the progression of the thesis work.

For the sake of completeness, it is fair saying that also an attempt with the Leaky ReLU activation function was performed, with slope equal to 0.0625, which is also supported by the NN compiler. The next table 4.6 compares these results with the ReLU case, showing that no significant improvement is given by its usage. Also considering that, the choice of the ReLU function seems to be reasonable.

	Easy dataset	Medium dataset	Hard dataset
MTCNN with ReLU	67.9%	66.5%	43.2%
MTCNN with Leaky ReLU	68.1%	66.7%	43.1%

Table 4.6. Comparison on the mean accuracy between the MTCNN with Leaky ReLU and ReLU activation functions.

Chapter 5

Hardware Implementation

In this chapter, changes made for the hardware implementation are explained and justified. Then, these simplifications are tested on the software implementation of the model to show the effects on the model's accuracy. Finally the resource usage and the timing analysis of the actual implementation are discussed.

5.1 Pre-processing

Pre-processing includes all operations done on the data before going to the actual computation with the CNN core module. As explained in previous section 4.2.1, the input image has a generic size and is rescaled several times. Each rescaled image is then fed inside the P-Net assuring the detection of faces at different scales. While this is a simple task to be implemented in software, especially with high level languages such as Python, it is not true when coding with VHDL. In parallel, has to been kept in mind that the final implementation, in principle, should work in real-time, so high complexity operations that lead to high accuracy are substituted with simpler ones that generates acceptable results.

The input images come from the integrated camera sensor on the top of the EVDK, properly set by writing on its internal registers. In order to save memory, frames from the camera have a fixed dimension of 160x120 pixels. This size can be obtained by the subsequent application of operations of sub-sampling, cropping and scaling on the primary exposed region where the Complementary MOS (CMOS) cells are placed. The work was speeded up thanks to an already existing Verilog code [30] for setting the camera, available on the open-source demos provided for the EVDK. So, the code was just adapted for this task. The next figure 5.1 summarized the steps performed.

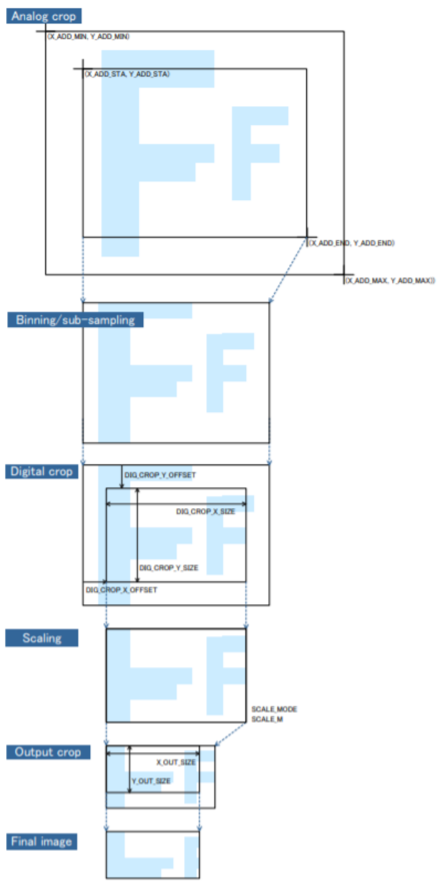


Figure 5.1. Image resizing in the camera module. Adapted from [3].

The format of the data at this point is the RAW10 format, where information for each pixel is embedded in 10 bits. In order to perform successive operations, the image must be converted on standard RGB format. Also in this case, an existing module from a demo [30] is used for the task, generating the three channels' data, with a width of 16 bits. Then, the data is saved on three dual-port RAM memories. It is important to notice that is not possible to avoid the usage of these memories, because the input image has to be accessible for all the successive stages.

The scaling operation for the saved input frame is performed with the set of fixed scaling factors [0.5 0.25 0.125]. These values are extremely easy to be implemented on hardware, in fact in order to obtain the scaled image, it is just necessary to properly select a smaller ensemble of the total pixels. In particular, for example in the case of a factor of 0.5, alternate rows and columns are taken from the input pixel matrix. The same is done in the cases of 0.25 and 0.125, where are not taken 3 and 7 rows (columns) in between two selected rows (columns), respectively. The reasoning on doing in this way is that near pixels in images have similar values in intensity, so the loss of information is limited. The next figure 5.2 shows the procedure adopted for a scaling factor of 0.5. On hardware it can be simply done by using a counter that increases with different steps depending on the needs.

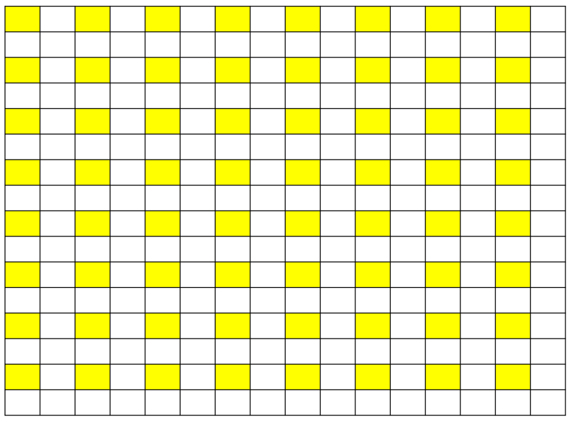


Figure 5.2. Rescaling procedure with a factor of 0.5.

To end this section about pre-processing, a black-box schematic of the implementation is given (fig. 5.3):

1. The 160x120 input frame is acquired by the camera sensor,
2. RGB values for each pixel are saved on three RAM memories,
3. A counter generates the correct addresses for the image RAMs in order to perform the rescaling,
4. Each scaled image is fed inside the CNN core for computation.

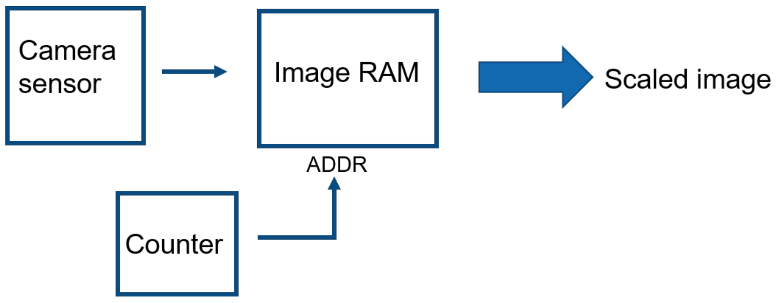


Figure 5.3. Schematic of the hardware implementation of the pre processing stage.

5.2 MTCNN

5.2.1 P-Net

The software implementation of the first stage has been described in details in 4.2.1, here only the changes made for its hardware implementation are discussed.

As mentioned in section 4.3.2, the NN compiler does not support the application of the softmax function for the classification output. In this way the detection output does not contain probability values in range $[0, 1]$ but numbers in the range $[-\infty, +\infty]$. While it is not a big problem in software, because the softmax can be applied outside the network for the successive calculations, in hardware it is not the way how it is done. One possible solution could be the application of the inverse function of the softmax to the thresholds and perform the comparison with values modified in the range $[-\infty, +\infty]$, but the problem is that the inverse of the softmax is not unique and the constant value to be added depends on its input data and varies each time it is applied to different inputs. This makes it not suitable for an hardware implementation. The solution adopted is to compare directly the face and no-face outputs from the CNN Accelerator Core, that is equal to use a threshold of 0.5 to which compare the face probability.

The second big simplification is the non implementation of the NMS step after CNN calculation. It has not be done because, although it can be performed, an optimized implementation of it on FPGA systems is a fairly recent field of study, see for example [17]. In order to avoid having a module that employs too much memories and that is not optimal, the final decision was to not implement it at this point and to consider all the proposals that pass the comparison stage.

The step performed in the first stage can be summarized as:

1. The scaled image is saved inside the CNN Accelerator Core, starting from the red channel and then blue and green ones. Each channel pixel intensity is represented with signed 16-bits. Correct addresses for internal memories of CNN Core have to be given in input.

2. Calculation with the CNN core is performed. Before asserting the starting signal, the initial address of the P-Net command code, previously saved on the DDR DRAM, has to be provided in input.

3. When o_we is asserted, the result data is collected. The data width is 16-bits. Firstly all the no-face results are serially given in output and saved on a RAM memory.

4. The next set of results is the face ones. Each single data coming from the CNN core is compared to its corresponding no-face result, previously saved on memory. The addresses of regions for which the condition $data_{face} \geq data_{no-face}$ is satisfied are saved, along with the corresponding face scores.

5. Then, the Bounding Box regression results are given in output. Just the data that correspond to the previously saved addresses is taken and, thanks to the knowledge of the address and of the scaling factor, the box coordinates are generated and calibrated. They are then saved on 4 different RAM memories, one for each X_1, Y_1, X_2, Y_2.

6. The process is repeated for each scaled image, X_1, Y_1, X_2 and Y_2 coordinates are saved on the corresponding RAM memories starting from the first available location.

7. At the end of the process, all the bounding box coordinates and relative scores for every scaled image are available into the corresponding memories.

The figure 5.4 shows a schematic of the process described above.

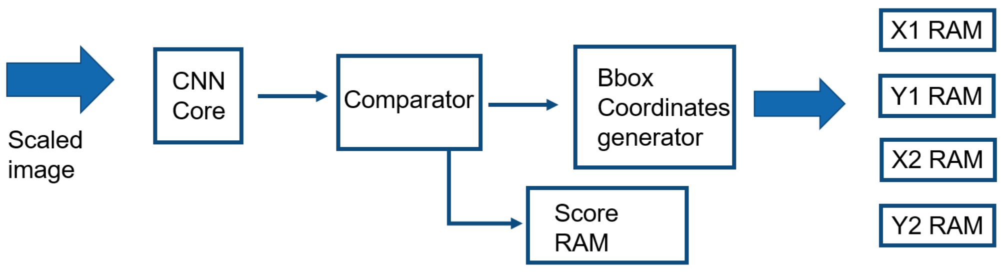


Figure 5.4. Schematic of the hardware implementation of the P-Net stage.

5.2.2 R-Net and O-Net

For the last two stages, the two simplifications made for the P-Net are still valid and they are:

- the comparison between the face and no-face outputs, instead of comparing the first one with a given threshold expressed as a probability and,
- the non implementation of NMS step.

Operations performed during second and third stages are essentially the same as explained in section 4.2.1. Here, the steps performed in the the hardware implementation are reported and a schematic high level block diagram is shown in fig. 5.5.

1. The coordinates of the proposal are taken from the X_1, Y_1, X_2 and, Y_2 RAMs. These ones are used for the generation of correct addresses for the image RAM, where the input frame is saved, in order to select the correct pixels inside the proposal region and perform the rescaling operation. The scaling operation is performed in a similar way as in section 5.1.

2. Pixels from the image RAM are saved in the internal memories of the CNN core, providing, also in this case, the correct addresses.

3. Computation of the CNN core is performed. The starting address of the external DDR DRAM, where the R-Net or O-Net command code is saved, must be provided.

4. When the o_we signal is high the output from the CNN module is collected. The first data is the no-face result that is saved in a 16-bits register.

5. The face result is compared to the no-face result, in the case of $data_{face} \geq data_{no-face}$ the coordinates are updated, otherwise the system waits until the CNN core is ready to accept the next proposal and the process restart from step 1.

6. The bounding box regression outputs ΔX_1 ΔY_1 ΔX_2 ΔY_2 are saved into 4 registers.

7. The coordinates X_1, Y_1, X_2, Y_2 are calibrated and saved back into X_1, Y_1, X_2 and, Y_2 RAMs. Then the next proposal is taken and the process is repeated.

8. The process finishes when all the proposals from the previous stage have been analysed. At the end of the O-Net stage the final results are saved in the X_1, Y_1, X_2 and, Y_2 RAM memories.

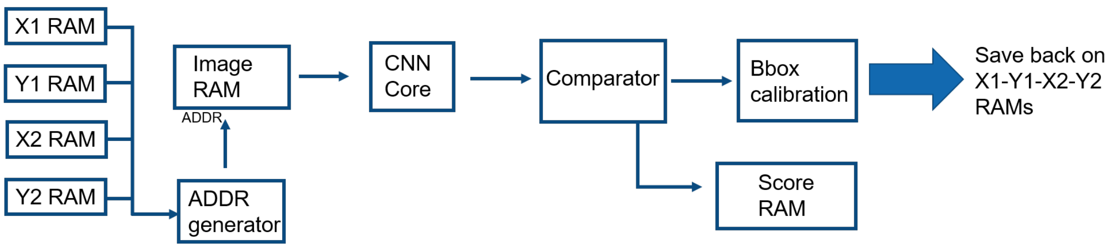


Figure 5.5. Schematic of the hardware implementation of R-Net and O-Net stages.

5.3 Accuracy Evaluation

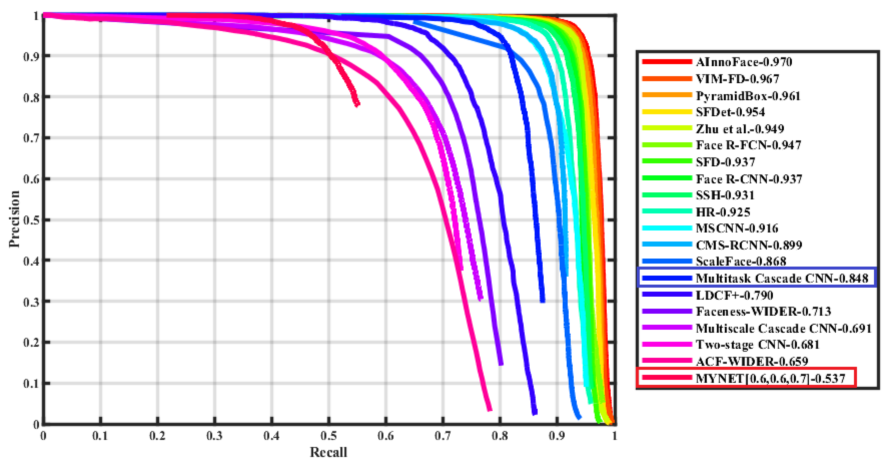
In this section, the effect on the accuracy of the model, because of the previous simplifications applied for the hardware implementation, are shown.

5.3.1 Effect of Further Simplifications

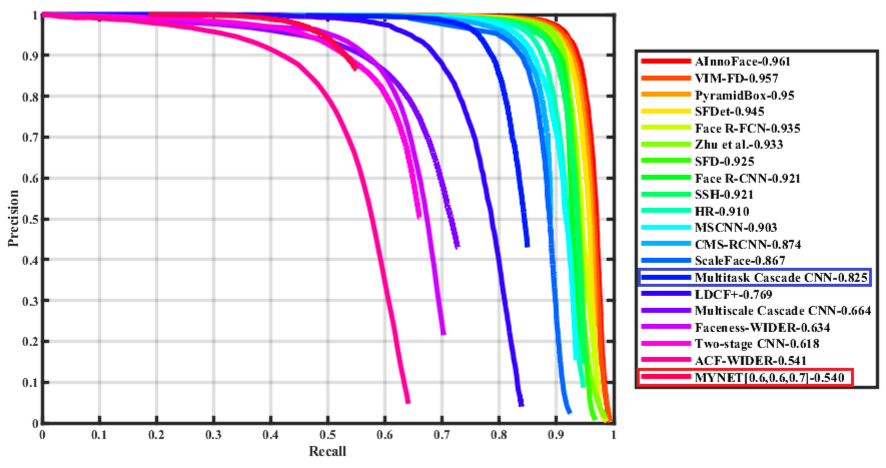
As discussed in section 4.2.2, the first changes made regard the elimination of the facial landmark localization output and the usage of ReLU activation function instead of the PReLU one.

Starting from this model, the successive modification applied is the usage of fixed scaling factors 0.5, 0.25 and 0.125. The successive fig. 5.6 shows the PR curves for the new model.

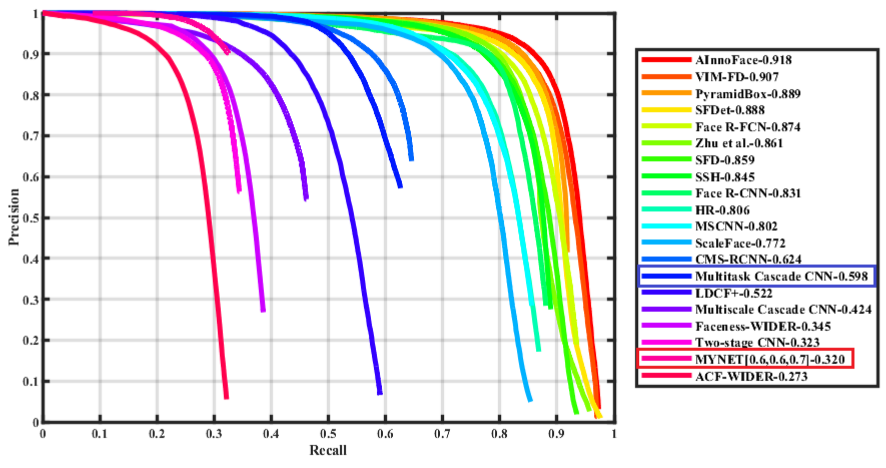
The last simplification concerns the thresholds after each stage. As explained in previous section 5.2, in hardware the comparison between the face result and the no-face one is performed for each proposal. In the software implementation, this is reflected in the usage of a threshold equal to 0.5 after each stage. The next figure 5.7 shows the PR curves for the model with also this change applied.



(a) Easy Dataset

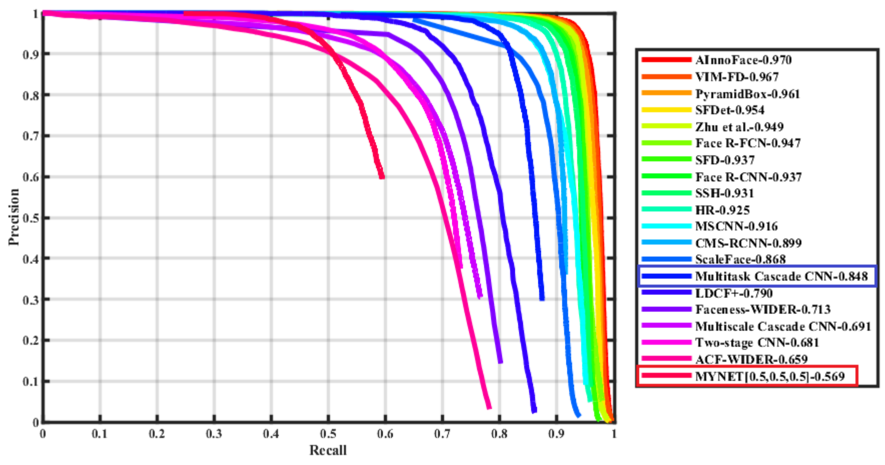


(b) Medium Dataset

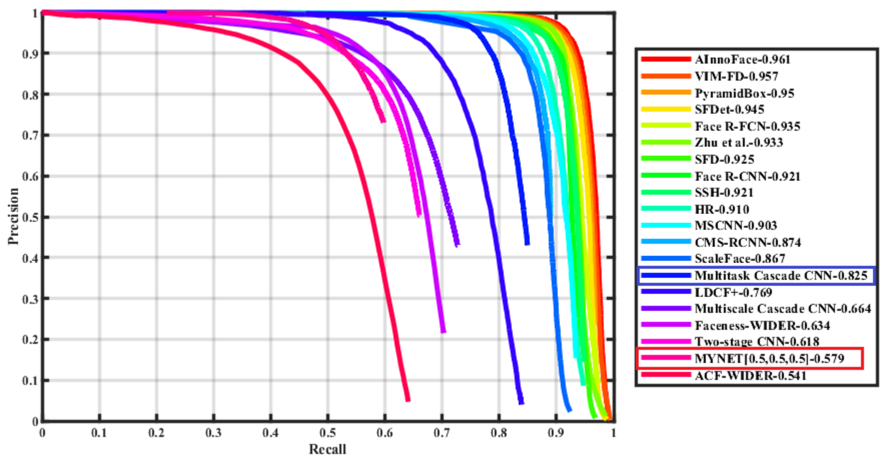


(c) Hard Dataset

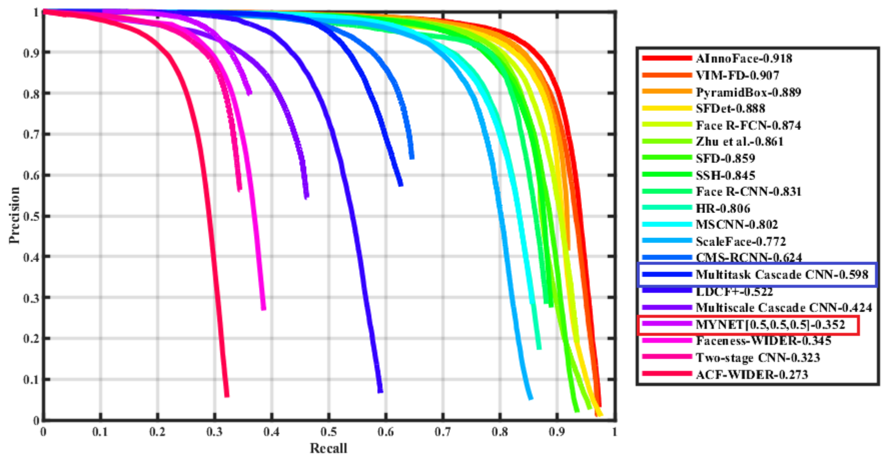
Figure 5.6. PR curves for the MTCNN model with fixed scaling factors.



(a) Easy Dataset



(b) Medium Dataset



(c) Hard Dataset

Figure 5.7. PR curves for the MTCNN model with fixed scaling factors and thresholds equal to 0.5.

The next table 5.1 compares the mean accuracy of the different models.

	Easy dataset	Medium dataset	Hard dataset
MTCNN with ReLU	67.9%	66.5%	43.2%
MTCNN with fixed scaling factors	53.7%	54.0%	32.0%
MTCNN with thresholds = 0.5	56.9%	57.9%	35.2%

Table 5.1. Comparison on the mean accuracy between the MTCNN from section 4.3.2 and the ones with fixed scaling factors and thresholds = 0.5.

As expected the accuracy of the model decreased from the case with an input image pyramid, because now some faces in images are no more detected. Clearly, this happens because the model is analysing three scaled images, and only faces at these scales are now detected. This generates the largest decrease in the mean accuracy of the model.
On the other hand, the application of a threshold equal to 0.5, slightly increases the mean accuracy of the model, because now more proposals than before are taken in early stages. This obviously increases the number of false negatives but also the faces with low face probability are now included, while in the previous models they were discarded.
The results are also comparable with what found in literature [1], that shows mean average precision values between 45.67% and 56.10% for the MTCNN, when using as input image a scaled version of the original one. The difference is that now the model can be successfully fully implemented on FPGAs and not only on CPUs or GPUs.

5.3.2 Effect of Quantization

The last evaluation about the accuracy regards the change from a 32-bits floating point model to a one that will perform calculations on 16-bits fixed point numbers. Usually, the process of reducing the number of bits for representing the weights of the model and for the calculations performed is called *quantization*. The comparison between the two models can be performed directly with the NN compiler. It provides the values inside each layer of the model for the floating point, fixed point and actual inference engine model, given a sample image at the input. The next figure 5.8 shows the comparison between the three models for the output of the O-Net.

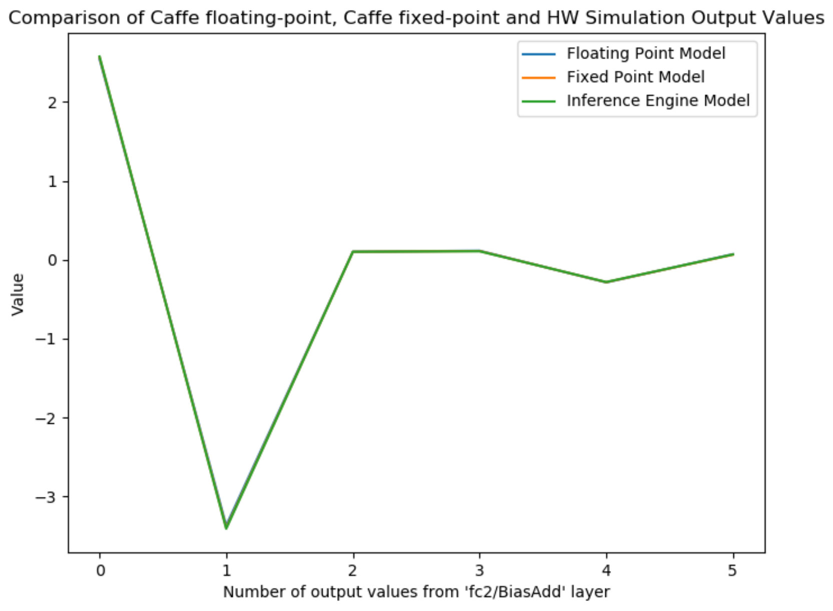


Figure 5.8. Comparison of the floating point, fixed point and actual hardware implementation results for the O-Net.

The graph represents the value for each of the six outputs coming from the O-Net in the case of different representation of numbers. In the graph, only the curve for the inference engine model is visible because the other two essentially overlap one to each other with the green one. This happens because the difference in the representation of numbers is very small, as shown in the next table 5.2 where the actual values for the three models are reported. In order to have a quantitative estimation of the error due to the quantization, the Mean Absolute Error (MAE) has been calculated, leading to a value of

$$MAE \simeq 0.014$$

	No-face output	Face output	ΔX_1	ΔY_1	ΔX_2	ΔY_2
Floating point model	2.544	−3.664	0.101	0.114	−0.288	0.0699
Fixed point model	2.564	−3.396	0.100	0.110	−0.285	0.0640
Hardware implementation	2.573	−3.406	0.101	0.107	−0.282	0.0659

Table 5.2. Actual values at the output of O-Net for floating point, fixed point and inference model.

5.4 Resource Usage

In this section the resource usage of the actual implementation of the face detector is discussed. As said in section 3.2, the target FPGA is the Lattice ECP5UM-85.

Before starting the actual development of the VHDL code for the system implementation, an estimation of the resource usage has been done. In particular, the focus is more on the memory usage, because it is limited to only 208 Embedded Block RAM (EBR), while for LUTs and register their number is quite high so that no particular interest has been given to them.

Firstly, the number of EBRs needed for the pre- and post-processing steps has been evaluated. The highest contribution comes from the memories needed for saving the input image and the results after the CNN core. Their number has been estimated to be about 60.
Considering the whole system, the CNN accelerator module is the one that uses most of the EBRs. Configuring it with the maximum computational capability with 8 engines leads to an amount of 153 EBRs for it. Considering the previous estimation for the rest of the system, this implementation is not suitable because exceeds the 208 EBRs available on the FPGA. So, it forces to a reduction of the computational power of the CNN core, that has been chosen to work with 6 engines leading to a number of EBRs equal to 117. In this way the system should fit into the target FPGA without any problem.
The actual resource usage for the hardware implementation of the face detector, after the Place&Route step, is reported in table 5.3.

EBRs	LUTs	1-bit Registers
207 (100%)	33638 (40%)	21005 (25%)

Table 5.3. Resource usage for the hardware implementation of the face detector after Place & Route step.

It shows that also with the CNN core working with 6 engines, almost 100% of the EBRs are used. This because the previous estimation for the pre- and post-processing was wrong. In fact, when performing the estimation, some modules were not considered and have been generated during the development of the VHDL code for the system. Essentially, as stated in sec. 5.1, between the camera sensor and the image RAM , a module for converting the input RAW10 data to RGB one is placed, and it uses a FIFO memory inside it. In addition, all the no-face results must to be saved after the CNN core and another memory is needed to save the addresses of regions for which $data_{face} > data_{no-face}$. Because of that the number of EBRs has increased, however the implementation is sutable for the FPGA.

5.5 Timing analysis

Also for what concerns the timing, some previous analysis have been done before starting with the actual implementation of the system.

Firstly the evaluation of the inference time per image for the software implementation of the MTCNN detector, including all the simplifications needed for its hardware counterpart, is reported. In this case, the model has been tested with images of the WIDER-FACE validation dataset, reporting a mean inference time per image equal to

$$\text{Inference time} \simeq 0.246\,s$$

showing that at least the software implementation on an Intel Core i7-9750H can be suitable for real-time detection with a throughput of about 4 Frame Per Second (FPS).

The second analysis has been done considering more the actual hardware setup used for the implementation. In this case the NN compiler has been used. After the compilation of the network, the number of cycles needed for a single computation is provided in output from the software and they are used for the estimation. In order to produce an estimation, the number of proposals coming from the first two steps has to be known. When doing this, the software implementation of the face detector was not working so a number of 100 proposals after the P-Net and 10 proposals after the R-Net were considered as feasible. The working frequency for the clock signal connected to the CNN core has been supposed to be 100 MHz. At this point, the number of clock cycles used for the pre- and post-processing of the data has not been considered because the main bottleneck for the computation is for sure the CNN core. The next table 5.4 reports the number of clock cycles for each of the three steps of the detection.

	P-Net	R-Net	O-Net	Total
clock cycles	932020	366505	1950475	3249001

Table 5.4. Number of clock cycles for the analysis of a single frame by the CNN core.

The result from these considerations is that the inference time for a single input frame is

$$\text{Inference time} \geq 0.57\,s$$

making it a good starting point for successive development and optimization of the face detector.

After the development of the VHDL code, the synthesis and the P&R steps the final outcome for the timing analysis has been given. It shows that

$$\text{Inference time} \geq 12.5\, s$$

making it not suitable for real time analysis at this point. The reasons behind this result are essentially two:

1. previously wrong estimation of the proposals coming from the first two stages. In fact, having at this point a working software implementation, a more accurate estimation has been done by using the number of proposals generated by the software implementation of the face detector with all the simplifications made previously. It shows that the mean number of proposals, by using the validation dataset from WIDER-FACE, is 870 and 120 after P-Net and R-Net respectively. The next table 5.5 reports, just for the sake of completeness, the number of clock cycles needed for each stage considering also the pre- and post-processing steps.

	Pre-processing	CNN core calculation	Post-processing
P-Net	18900 clk cycles	759196 clk cycles	6240 clk cycles
R-Net	1503360 clk cycles	313880340 clk cycles	330000 clk cycles
O-Net	794880 clk cycles	207462300 clk cycles	3750 clk cycles

Table 5.5. Number of clock cycles for the analysis of a single frame by the system implemented.

2. limited working frequency for the CNN accelerator core. After the P&R step the maximum frequency for the clock connected to the CNN core is limited to 42 MHz.

At this point, it is important to know which is the logical path that is limiting the working frequency of the system. It has been discovered that the critical path is the one for the generation and calibration of bounding boxes coordinates after the P-Net computation, as highlighted in the next fig. 5.9.

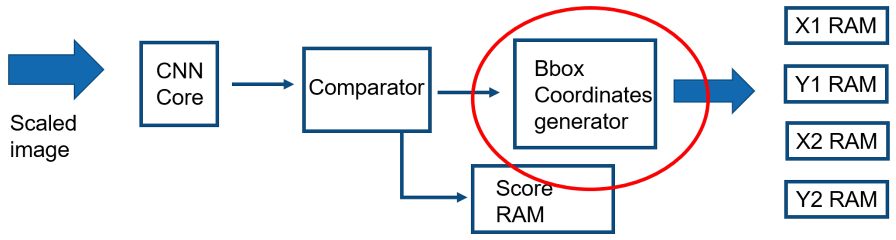


Figure 5.9. Schematic of the P-Net stage with the highlighted critical path.

The reason is that for this block, the operations have been done with the help of variables in VHDL, leaving the problem of how to convert the operations into logical modules to the synthesis tool. It has been done in this way because the entries of the output tensor are given in a serial way and are also saved serially on memories. So, from the address of the location where the data is saved, it is necessary to calculate the corresponding X_1, Y_1, X_2 and Y_2 for the proposal, in order to apply equations 4.1 and 4.2 and then equations 4.4 and 4.5. These calculations imply divisions and multiplications for numbers that are not power of two. For example, considering the output tensor for the first rescaled image, that has a dimension of 35x25x6, the X_1 coordinate for the proposal saved at a specific address can be calculated as:

$$num_{row} = (addr + 1)/25$$

$$num_{col} = X_1 = addr - (num_{row} \cdot 35)$$

At the time of the implementation a more optimal way was not found, so the usage of variables has been the chosen solution. In this way the division and multiplication operations have been synthesized with the usage of multiplier modules, that are by definition quite slow arithmetic modules.

One possible optimization can be the substitution of the multiplication operation with shift-left and add ones, for example

$$x \cdot 35 = x \cdot (32 + 3) = x \cdot (2^5 + 2 + 1) = (x << 5) + (x << 1) + x$$

where $<<$ represent the shift left operation.
The division can be substituted with a recursive subtraction of the divisor from the dividend: the number of subtractions that can be performed until $dividend \geq divisor$ is counted and it gives the solution of the division to be rounded depending on the value of the remainder. Due to a lack of time for the project, these optimizations have not been implemented because they require a change of the structure around where the operations are performed, but in principle they should lead to better performances thanks to the removal of multipliers modules.

Chapter 6

Conclusions and Future Outlooks

At the end of the experience, a fully integrated face detection algorithm has been developed for a low power FPGA, that was the primary task of the thesis. The steps performed can be summarized as:

- Study and identification of a suitable Deep Learning approach for face detection problem that can fit in a low power FPGA with limited memory and resources.
- The chosen method, the MTCNN network, has been implemented on software and trained. Several simplification from the initial model have been applied in order to make it suitable for the successive hardware integration. The most important ones are the removal of the facial alignment output and the fixed scaling factors for the pre-processing of the input frame. The effects of the simplifications on the accuracy of the model have been discussed showing that the highest impact on the loss of accuracy is due to the previously two cited changes.
- An high level structure of the system implemented has been developed for a previous estimation of resource usage and for expectations on the timing analysis. Then, with a top-down approach, the VHDL code for the face detector has been developed, almost from the beginning, and the synthesis and P&R steps have been performed in order to show that it can be integrated into the target low power FPGA.

Although the final solution has an inference time that makes it not suitable for a real-time application, the approach adopted for its implementation is completely new and introduces new challenges. Usually, only the CNN accelerator module is developed by using FPGAs [5], creating a module optimized for the specific network, leaving the pre- and post-processing steps on CPUs or GPUs that have less computation limitations but higher power consumptions than a FPGA. In this work, also the last two mentioned tasks have been implemented on FPGA, making necessary simplifications and solutions for problems arisen during the translation of operations from software to hardware.

In conclusion, it can be certainly used a starting point for successive development of CV applications that can perform calculations directly on sensing devices. Next steps for the project could be:

- Development of a complete testbench for the whole system to ensure the correct timing interface between different modules. At this point only developed modules have been tested singularly.
- Reduction of inference time per frame, by acting in two ways:
    1. reduction of the number of proposals from the two early stages, acting both on the training process of the network and with the integration on hardware of some filtering modules as it is done on software with the NMS algorithm;
    2. optimization of the critical path that limits the working frequency. Instead of using variables and leaving the synthesis of logical blocks to the tolls, probably a by hand declaration of the arithmetic modules should lead to better performances. The usage of some pipelining registers could be another good solution.
- Optimal power analysis should be done by loading the code on the FPGA and measuring the consumptions during computation thanks of a power analyser. It is an essential step for a portable application.

Bibliography

[1] D. Chaves et al. "CPU vs GPU performance of deep learning based face detectors using resized images in forensic applications". In: *9th International Conference on Imaging for Crime Detection and Prevention (ICDP-2019).* 2019, pp. 93–98.

[2] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers.* 2016, 262–263.

[3] Sony Semiconductor Corporation. *IMX214 Software Reference Manual.* Version 4.1.3. 2018.

[4] Xin Feng et al. "Computer vision algorithms and hardware implementations: A survey". eng. In: *Integration (Amsterdam)* 69 (2019), pp. 309–320. ISSN: 0167-9260.

[5] C. Fu and Y. Yu. "FPGA-based Power Efficient Face Detection for Mobile Robots". In: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO).* 2019, pp. 467–473.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[7] Richard Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405 (July 2000), pp. 947–51. DOI: `10.1038/35016072`.

[8] Simon Haykim. *Neural Networks and Learning Machines.* 3rd ed. Upper Saddle River, NJ: Pearson Education, 2009.

[9] K. He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision (ICCV).* 2015, pp. 1026–1034.

[10] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.* 2017. arXiv: `1704.04861 [cs.CV]`.

[11] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: 1502.03167 [cs.LG].

[12] J. Jo et al. "DSIP: A Scalable Inference Accelerator for Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 53.2 (2018), pp. 605–618.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[14] Anders Krogh and John A. Hertz. "A Simple Weight Decay Can Improve Generalization". In: *Advances in Neural Information Processing Systems 4*. Ed. by J. E. Moody, S. J. Hanson, and R. P. Lippmann. Morgan-Kaufmann, 1992, pp. 950–957. URL: http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf.

[15] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[16] Haoxiang Li et al. "A convolutional neural network cascade for face detection". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 5325–5334.

[17] F. Liang et al. "The Design of Objects Bounding Boxes Non-Maximum Suppression and Visualization Module Based on FPGA". In: *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP).* 2018, pp. 1–5.

[18] Ziwei Liu et al. "Deep Learning Face Attributes in the Wild". In: *Proceedings of International Conference on Computer Vision (ICCV).* Dec. 2015.

[19] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4(5) (1964), pp. 1–17.

[20] Dr. Adrian Rosebrock. *Deep Learning for Computer Vision with Python - Starter Bundle.* 1st ed. Vol. 1. PyImageSearch, 2017.

[21] M. Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2018, pp. 4510–4520.

[22] Lattice Semiconductor™. *CNN Accelerator IP Core - User Guide.* Version 2.1. Oct. 2019.

[23] Lattice Semiconductor™. *CrossLink VIP Input Bridge Board - Evaluation Board User Guide.* Version 1.4. Oct. 2018.

[24] Lattice Semiconductor™. *ECP5 and ECP5-5G Family - Datasheet.* Version 2.1. Apr. 2019.

[25] Lattice Semiconductor™. *ECP5 VIP Processor Board - Evaluation Board User Guide*. Version 1.4. Feb. 2018.

[26] Lattice Semiconductor™. *HDMI VIP Output Bridge Board - Evaluation Board User Guide*. Version 1.1. June 2017.

[27] Lattice Semiconductor™. *Lattice Embedded Vision Development Kit - User Guide*. Version 1.3. Nov. 2018.

[28] Lattice Semiconductor™. *Lattice Memory Mapped Interface and Lattice Interrupt Interface - User Guide*. Version 1.1. Feb. 2018.

[29] Lattice Semiconductor™. *Lattice SensAI Neural Network Compiler Software - User Guide*. Version 2.1. Oct. 2019.

[30] Lattice Semiconductor™. *Object Counting using Mobilenet CNN Accelerator IP - Reference Design*. Oct. 2019.

[31] Lattice Semiconductor™. *USB3-GbE VIP IO Board -Evaluation Board User Guide*. Version 1.0. May 2018.

[32] A. Shrivastava, A. Gupta, and R. Girshick. "Training Region-Based Object Detectors with Online Hard Example Mining". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2016, pp. 761–769. DOI: `10.1109/CVPR.2016.89`. URL: `https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.89`.

[33] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: `1409.1556 [cs.CV]`.

[34] SONY. *IMX214-0AQH5-C datasheet*.

[35] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. arXiv: `1412.6806 [cs.LG]`.

[36] Paul Viola and Michael Jones. "Robust Real-Time Face Detection". In: *International Journal of Computer Vision* 57 (May 2004), pp. 137–154. DOI: `10.1023/B:VISI.0000013087.49260.fb`.

[37] Shuo Yang et al. "WIDER FACE: A Face Detection Benchmark". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[38] Zhenheng Yang and R. Nevatia. "A multi-scale cascade fully convolutional network face detector". In: *2016 23rd International Conference on Pattern Recognition (ICPR)* (2016), pp. 633–638.

[39] Bin Zhang et al. *ASFD: Automatic and Scalable Face Detector*. 2020. arXiv: `2003.11228 [cs.CV]`.

[40] K. Zhang et al. "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks". In: *IEEE Signal Processing Letters* 23.10 (2016), pp. 1499–1503.

[41] Shifeng Zhang et al. *RefineFace: Refinement Neural Network for High Performance Face Detection*. 2019. arXiv: `1909.04376 [cs.CV]`.

Appendix A

Detailed Structure of the Hardware Implementation

In this appendix a more detailed description of the structure of the hardware implementation of the face detector is given, including the modules not discussed in the previous chapters because not fundamental for the understanding but present in order to ensure a correct functionality.

The system has been developed as an Algorithm State Machine with 102 states, in each state a specific operation of the data flow is performed. Due to the large number of operations and the complexity of the system, it has been chosen to use separate states for simple operations, in order to have a better control on the different modules. It can be done without any risk due to the large number of 1-bit registers available on the ECP5UM-85 FPGA.
The working process flow of the detector is described as follows:

1. The command code has to be generated by the NN compiler for the three networks and loaded on a SD card.

2. The SD card with the command code for the successive CNN core calculations has to be connected to the board and its content to be loaded on the integrated DDR3 RAM. It is performed in the early states of the process after the application of the reset signal. Practically, the data from the connected SD card is taken by a module called *sd_spi* with the aim to generate the signals for the AXI3 protocol for writing into the DDR3 RAM. These signals constitute a set of inputs for the *axi_ws2m1* module: it is an arbiter that controls which module can write into the DDR3 RAM. This is essential because also the CNN core needs to write into external memories, in this way one module at the time have writing access to the DDR3 RAM. The outputs from the *axi_ws2m1* are inputs for the *axi2lattice128* module that generates the data burst for the

ddr3_ctrl which actually writes and reads data from the integrated memories.

3. In parallel, the two external cameras must be set for having the correct format of frames. It is done thanks to the module *CSI2_to_DVI_top*, that consists of two sub-modules:
   - the first one is the *i2c_top* sub-module, it writes into the cameras' register in order to set them;
   - the second is the *image_pipe* one that takes the data from cameras in the RAW10 format and converts it to standard RGB data, with a width of 16 bits for each channel.
4. When the done signal for setting correctly everything is asserted, the system starts saving the input frame into the three image RAMs, one for each channel.
5. After the saving of the input frame, the data is written into the CNN Accelerator Core by using a counter for generating the correct internal memories addresses and the computation is performed, as detailed in sec. 3.2.1. During computation, the CNN core reads the command code from the external memory and writes partial data into it if its internal memories are not sufficient. For reading data from the DDR3 RAM, the CNN core AXI3 outputs are taken in input by the *axi2lattice* module that generates the correct set of signals to be sent to the *ddr3_ctrl* for the reading of data. For the writing of data the process is quite the same, but the request of writing has to pass on the arbiter *axi_ws2m1* before to be taken in input by the *axi2lattice* module.
6. After the computation of the CNN core, the data is processed and saved on the result memories, as explained in sec. 5.2.
7. When the last results are saved into memories, the process restarts waiting for the next complete frame to be analysed.

The modules *sd_spi*, *axi_ws2m1*, *axi2lattice128* and *CSI2_to_DVI_top* are taken from an existing demo [30], while the *ddr_ctrl* and all the memories used are generated through the diamond software for the generation of the IP, as well as for the CNN Accelerator Core. All the other modules, mostly different types of counters, have been described by hand and singularly tested.