UNIVERSITE DE BOURGOGNE

UFR Sciences Humaines

THÈSE
Pour obtenir le grade de

Docteur de l'Université de Bourgogne

Discipline : Informatique

par
Olivier Boisard

le 25 octobre 2016

Optimization and implementation of bio-inspired feature extraction frameworks for visual object recognition

Directeur de thèse
Michel Paindavoine

Jury
Philippe Coussy, Professeur des universités, Rapporteur
Christophe Garcia, Professeur des universités, Rapporteur
Andres Perez-Uribe, Professeur des universités, Examinateur
Robert M. French, Directeur de recherches au CNRS, Examinateur
Michel Doussot, Maître de conférences, Examinateur
Olivier Brousse, Ingénieur de Recherche, Examinateur

"Airplanes don't flap their wings and don't have feathers."

Yann LeCun

Abstract

Industry has growing needs for so-called "intelligent systems", capable of not only acquire data, but also to analyse it and to make decisions accordingly. Such systems are particularly useful for video-surveillance, in which case alarms must be raised in case of an intrusion. For cost saving and power consumption reasons, it is better to perform that process as close to the sensor as possible. To address that issue, a promising approach is to use bio-inspired frameworks, which consist in applying computational biology models to industrial applications. The work carried out during that thesis consisted in selecting bio-inspired feature extraction frameworks, and to optimize them with the aim to implement them on a dedicated hardware platform, for computer vision applications. First, we propose a generic algorithm, which may be used in several use case scenarios, having an acceptable complexity and a low memory print. Then, we proposed optimizations for a more global framework, based on precision degradation in computations, hence easing up its implementation on embedded systems. Results suggest that while the framework we developed may not be as accurate as the state of the art, it is more generic. Furthermore, the optimizations we proposed for the more complex framework are fully compatible with other optimizations from the literature, and provide encouraging perspective for future developments. Finally, both contributions have a scope that goes beyond the sole frameworks that we studied, and may be used in other, more widely used frameworks as well.

Résumé

L'industrie a des besoins croissants en systmes dits intelligents, capable d'analyser les signaux acquis par des capteurs et prendre une dcision en consquence. Ces systmes sont particulirement utiles pour des applications de vido-surveillance ou de contrle de qualit. Pour des questions de cot et de consommation d'nergie, il est souhaitable que la prise de dcision ait lieu au plus prs du capteur. Pour rpondre cette problmatique, une approche prometteuse est d'utiliser des mthodes dites bio-inspires, qui consistent en l'application de modles computationels issus de la biologie ou des sciences cognitives des problmes industriels. Les travaux mens au cours de ce doctorat ont consist choisir des mthodes d'extraction de caractristiques bio-inspires, et les optimiser dans le but de les implanter sur des plateformes matrielles ddies pour des applications en vision par ordinateur. Tout d'abord, nous proposons un algorithme gnrique pouvant tre utiliss dans diffrents cas d'utilisation, ayant une complexit acceptable et une faible empreinte mmoire. Ensuite, nous proposons des optimisations pour une mthode plus gnrale, bases essentiellement sur une simplification du codage des donnes, ainsi qu'une implantation matrielle bases sur ces optimisations. Ces deux contributions peuvent par ailleurs s'appliquer bien d'autres mthodes que celles tudies dans ce document.

Acknowledgements

So here I am, after three years spent playing around with artificial neurons. That went fast, and I guess I would have needed twice as long to get everything done. That was a great experience, which allowed me to meet extraordinary people without whom those years wouldn't have been the same.

First of all, I wish to thank my mentor Michel Paindavoine for letting me be his student, along with my co-mentors Olivier Brousse and Michel Doussot. All three showed real implication in my work, and your advices and the long discussions we had was of great help. I would also like to thank Xavier Bruneau, for allowing me to realize that thesis in his company – too often we hear about PhD students in companies who had no time for academic works, and I'm glad I wasn't one of them. I also wish to thank Bob French and Lionel Lacassagne for accepting to be part of the follow-up committee, as well as Thomas Serre and all the staff at Serre Lab for welcoming me. Then I would like to thank Philippe Coussy and Christophe Garcia, for being the first readers of this document and accepting to review it. Finally, I would like to thank the ANRT, i.e the French National Research and Technology Agency, for giving me the opportunity to realize that PhD with the CIFRE program.

But all of those years wouldn't have been the same without my colleagues and fellow PhD students, at GlobalSensing Technologies and at the LEAD. So I thank you all, whether you were there for a few months only or several years, for your support, advices, and the laughs that we shared: Mathieu, Lolita, Laura, Luc, Jonathan, Sabrina, Vivien, Rémi (*he ain't here!*), Pierre, Julie, Margaux, Stéphane, Alessandro, Danilo, Philippe, Corinne, Sandrine, Léa, Lydia, Christophe, Radka, Yannick, Alex, Axel, Guillaume, Guillaume, Éric, David, Christophe... And I sure hope I haven't forgotten anybody!

Finally, although working was my primary hobby during those years, I wouldn't have gone through that adventure without my relatives, friends and family. So I would like to thank my family, and particularly my parents Jean-Louis and Françoise, my adoptive parents Phi and Sun, my brother and sister Vincent and Marianne and their respective wife and husband Anastasia and Matthieu, and my adoptive brother Jérémy for their outstanding support despite the distance. I also want to thank my friends – some of them shared a bit of my life, in their own ways. So thank you to Benoît, Kiki, Drak, Émilie, Rémi "Goodfinger", Manjo, Élisa, Clémence, Romain, Roswitha, Hélène, Margot, Alena, Chloé, Jimmy, Mélissa, Valentin, David, Claire, Jordan, Alexis, Franck, Thomas "Vougny-Pensez-Pas", Manon, Jack, Annabelle, Matthieu, Yankel, Paméla, Céline, and Antoine.

Thank you all!

Contents

Abstract iii

Résumé iv

Acknowledgements v

List of Figures xii

List of Tables xiv

1 General introduction 1
- 1.1 The need for intelligent systems . . . 1
- 1.2 Machine Learning . . . 2
- 1.3 Embedded systems . . . 3
- 1.4 NeuroDSP: a neuro-inspired integrated circuit . . . 4
- 1.5 Document overview . . . 5

2 Related works and problem statement 7
- 2.1 Theoretical background . . . 7
  - 2.1.1 Classification frameworks . . . 7
    - 2.1.1.1 Neural Networks . . . 8
      - Perceptron . . . 9
      - Multilayer Perceptron . . . 10
      - RBF . . . 11
      - Spiking Neural Network . . . 12
    - 2.1.1.2 SVM . . . 13
    - 2.1.1.3 Ensemble learning . . . 13
  - 2.1.2 Feature extraction frameworks . . . 14
    - 2.1.2.1 Signal processing approach . . . 14
      - Classical approaches . . . 14
      - Wavelets . . . 16
    - 2.1.2.2 A biological approach: HMAX . . . 17
    - 2.1.2.3 ConvNet . . . 19
- 2.2 Frameworks implementations . . . 20

2.2.1 Software implementations . . . 20
2.2.1.1 Workstations . . . 21
2.2.1.2 Embedded systems . . . 21
2.2.2 Hardware implementations . . . 21
2.2.2.1 Neural networks . . . 22
HMAX . . . 22
ConvNet . . . 24
Spiking Neural Networks . . . 26
2.2.2.2 Other frameworks implementations . . . 27
2.3 Discussion . . . 29
2.3.1 Descriptors and classifiers comparison . . . 29
2.3.1.1 Descriptors . . . 29
2.3.1.2 Classifiers . . . 31
2.3.2 Implementations comparison . . . 32
2.3.3 Problem statement . . . 33
2.4 Conclusion . . . 33

3 Feature selection **35**
3.1 Feature selection for face detection . . . 35
3.1.1 Detecting faces . . . 35
3.1.1.1 Cascade of Haar-like features . . . 36
Framework description . . . 36
Complexity analysis . . . 37
Memory print . . . 40
3.1.1.2 CFF . . . 41
Framework description . . . 41
Complexity analysis . . . 41
Memory print . . . 44
3.1.1.3 HMIN . . . 45
Framework description . . . 45
Complexity analysis . . . 45
Memory print . . . 46
3.1.2 HMIN optimizations for face detection . . . 46
3.1.2.1 C1 output . . . 46
3.1.2.2 Proposed optimizations . . . 47
$\text{HMIN}_{\theta=\pi/2}$ . . . 47
$\text{HMIN}^{R}_{\theta=\pi/2}$ . . . 48
3.1.3 Experiments . . . 49
3.1.3.1 Test on LFWCrop_grey . . . 49
3.1.3.2 Test on CMU . . . 50
3.1.3.3 Test on Olivier dataset . . . 54
3.2 Feature selection for pedestrian detection . . . 55
3.2.1 Detecting pedestrians . . . 55
3.2.1.1 HOG . . . 56
Gradients computation . . . 56
Binning . . . 57
Local normalization . . . 58

Complexity analysis . . . 59
Memory print . . . 60
3.2.1.2 ConvNet . . . 61
Presentation . . . 61
Complexity analysis . . . 62
Memory print . . . 65
3.2.2 HMAX optimizations for pedestrian detection . . . 66
3.2.3 Experiments . . . 66
3.3 Discussion . . . 66
3.4 Conclusion . . . 69

4 Hardware implementation **71**
4.1 AAM for HMAX . . . 71
4.1.1 Description . . . 72
4.1.1.1 S1 . . . 72
4.1.1.2 C1 . . . 73
4.1.1.3 S2 . . . 73
4.1.1.4 C2 . . . 74
4.1.2 Results . . . 74
4.2 Proposed simplification . . . 74
4.2.1 Input data . . . 76
4.2.2 S1 filters coefficients . . . 77
4.2.3 S1 output encoding . . . 78
4.2.4 Filter reduction in S2 . . . 81
4.2.5 Manhattan distance in S2 . . . 82
4.3 FGPA implementation . . . 82
4.3.1 Overview . . . 82
4.3.2 `s1c1` . . . 84
4.3.2.1 `s1` . . . 84
`pixel_manager` . . . 85
`pix_to_stripe` . . . 85
`pixmat` . . . 86
`coeffs_manager` . . . 86
`conv_filter_bank` . . . 86
`conv_crop` . . . 87
4.3.2.2 `conv` . . . 88
`convrow` . . . 88
`sum_acc` . . . 88
`s1degrader` . . . 89
4.3.2.3 `shift_registers` . . . 90
4.3.2.4 `c1` . . . 90
`c1_max_2by2` . . . 91
`c1_pix_to_stripe` . . . 91
`c1_reorg_stripes` . . . 92
`c1_orientations_demux` . . . 92
`c1_orientation` . . . 92
`c1unit` . . . 92

`maxfilt` . . . 93
`c1unit_ctrl` . . . 94
4.3.2.5 `c1_to_s2` . . . 94
`c1_handler` . . . 95
4.3.3 `s2c2` . . . 97
4.3.3.1 `s2` . . . 97
4.3.3.2 `s2_input_manager` . . . 98
`s2_input_handler` . . . 99
`s2_pix_to_stripe` . . . 101
4.3.3.3 `s2_coeffs_manager` . . . 101
4.3.3.4 `s2processors` . . . 101
4.3.3.5 `corner_cropper` . . . 102
4.3.3.6 `s2bank` . . . 102
4.3.3.7 `s2unit` . . . 103
`cum_diff` . . . 104
4.3.4 `c2` . . . 104
4.3.4.1 `c2_to_out` . . . 105
4.4 Implementation results . . . 105
4.4.1 Resource utilization . . . 105
4.4.2 Timing . . . 106
4.5 Discussion . . . 107
4.6 Conclusion . . . 109

5 Conclusion **111**

A RBF networks training **116**
A.1 Overview . . . 116
A.2 Clustering . . . 117
A.3 Output layer training . . . 118

B Résumé en français **119**
B.1 Introduction générale . . . 119
B.2 État de l'art . . . 121
B.2.1 Fondements théoriques . . . 121
B.2.1.1 Méthodes de classification . . . 121
B.2.1.2 Méthodes d'extraction de caractéristiques . . . 122
B.2.2 Implantations matérielles . . . 125
B.2.3 Discussion . . . 126
B.3 Sélection de caractéristiques . . . 127
B.3.1 Détection de visages . . . 127
B.3.1.1 Viola-Jones . . . 127
B.3.1.2 CFF . . . 128
B.3.1.3 HMIN et optimisations . . . 129
B.3.1.4 Expérimentations . . . 129
B.3.2 Détection de piétons . . . 131
B.3.2.1 HOG . . . 132

B.3.2.2 ConvNet . . . 132
B.3.2.3 Expérimentations . . . 133
B.3.3 Conclusion . . . 133
B.4 Implantation matérielle . . . 134
B.4.1 Optimisations . . . 135
B.4.1.1 Données en entrée . . . 135
B.4.1.2 Filtres de Gabor . . . 135
B.4.1.3 Autres optimisations . . . 136
B.4.2 Résultats d'implantation . . . 137
B.4.3 Conclusion . . . 138
B.5 Conclusion . . . 139

Publications **140**

Bibliography **141**

List of Figures

1.1 Application examples . . . 2
1.2 Perceptron applied to PR . . . 3
1.3 NeuroDSP architecture. . . . 5

2.1 A feedforward architecture . . . 9
2.2 Perceptron. . . . 11
2.3 Multi-layer perceptron . . . 11
2.4 MLP activation functions. . . . 12
2.5 RBF neural network. . . . 13
2.6 Support vectors determination . . . 14
2.7 Invariant scattering convolution network. . . . 16
2.8 HMAX. . . . 20
2.9 Convolutional neural network. . . . 20

3.1 Example of Haar-like features used in Viola-Jones. . . . 37
3.2 Integral image representation. . . . 38
3.3 Complexity repartition of Viola and Jones' algorithm. . . . 40
3.4 CFF. . . . 42
3.5 Complexity repartition of the CFF algorithms . . . 44
3.6 C1 feature maps for a face . . . 47
3.7 S1 convolution kernel sum . . . 48
3.8 Feature map obtained with the unique kernel in S1 . . . 48
3.9 ROC curves of the HMIN classifiers. . . . 51
3.10 Samples from the CMU Face Images dataset . . . 52
3.11 ROC curve obtained with $\text{HMIN}^{R}_{\theta=\pi/2}$ on CMU dataset. . . . 53
3.12 Example of frame from the "Olivier" dataset. . . . 54
3.13 ROC curves obtained with $\text{HMIN}^{R}_{\theta=\pi/2}$ on "Olivier" dataset. . . . 55
3.14 HOG descriptor computation. . . . 57
3.15 Binning of the half-circle of unsigned angles . . . 58
3.16 Complexity repartition of HOG features extraction. . . . 60
3.17 ConvNet for pedestrian detection. . . . 62
3.18 ROC curves of the HMIN classifiers on the INRIA pedestrian dataset . . . 67

4.1 Caltech101 samples . . . 75
4.2 Precision degradation in input image. . . . 76
4.3 Recognition rates of HMAX w.r.t input image bit width. . . . 77
4.4 Recognition rates w.r.t S1 filters precision . . . 79
4.5 HMAX VHDL module . . . 84

4.6 Dataflow in `s1`. . . . 85
4.7 `coeffs_manager` module. . . . 87
4.8 7×7 convolution module. . . . 89
4.9 `shift_registers` 90
4.10 `c1` module 91
4.11 `c1unit` 93
4.12 `c1_to_s2` module 96
4.13 Dataflow in `s2c2`. The data arriving to the module is handled by `s2_input_manager`, which make it manageable for the `s2processors`. The latter also gets the pre-learnt filter needed for the pattern-matching operations from `s2_coeffs_manager` in parallel, and perform the computations. Once it is over, the data is sent in parallel to the `dout` output port, which feed the next processing module. . . . 98
4.14 Data management in `s2_handler`. . . . 100
4.15 Data flow in `s2processors`. . . . 103

B.1 Exemples d'applications 120
B.2 NeuroDSP architecture. . . . 120
B.3 Architecture *feedforward* 122
B.4 *Invariant scattering convolution network*. . . . 123
B.5 HMAX. . . . 124
B.6 Réseaux de neurones à convolutions. . . . 125
B.7 Examples de caractéristiques utilisés dans Viola-Jones. . . . 127
B.8 Représentation en image intégrale. . . . 128
B.9 CFF. . . . 128
B.10 Sorties des C1 pour un visage 130
B.11 Somme des noyaux de convolutions dans S1. . . . 130
B.12 Réponse du filtre unique dans S1 sur un visage. . . . 130
B.13 Courbes ROC obtenues avec différentes versions de HMIN sur LFW_Crop. 131
B.14 Courbe ROC obtenue avec $\text{HMIN}^{R}_{\theta=\pi/2}$ sur la base CMU. . . . 131
B.15 HOG 132
B.16 ConvNet pour la détection de piétons. . . . 133
B.17 Courbes ROC obtenues avec les descripteurs HMIN sur la base INRIA. . . 133
B.18 Effet de la dégradation de précision sur l'image d'entrée. . . . 135
B.19 Taux de reconnaissances avec HMAX en fonction de la précision des pixels en entrée. . . . 136
B.20 Précisions en fonction du nombres de bits dans les filtres de Gabor de S1, avec 2 bits pour l'image d'entrée. . . . 137
B.21 Aperçu du module VHDL HMAX. . . . 137

List of Tables

2.1 Paramaters for HMAX S1 and C1 layers 19
2.2 Comparison of descriptors. 30

3.1 Accuracies of the different version of HMIN on the LFW_crop dataset. . . 50
3.2 Complexity and accuracy of face detection frameworks. 53
3.3 Complexity and accuracy of human detection frameworks. 67

4.1 Hardware resources utilized by Orchard's implementation 74
4.2 Accuracies of Orchard's implementations on Caltech101. 75
4.3 Code books and partitions for C1 81
4.4 Accuracies of HMAX with several optimizations. 83
4.5 Address offsets in `c1_to_s2` 96
4.6 Mapping between N and `dout_scale`. 99
4.7 Resource utilization of HMAX implementation on XC7A200TFBG484-1. . 106
4.8 Hardware resources comparison between the Virtex-6 FPGA used in [99], and the Artix-7 200T we chose. 109

B.1 Paramètres des couches S1 et C1 de HMAX 125
B.2 Comparaison des principaux extracteurs de caractéristiques. 126
B.3 Précision des différentes versions de HMIN sur la base de données LFW_crop. 129
B.4 Complexité et précision de différentes méthodes de détections de visages . 132
B.5 Complexité et précisions de différentes méthode de détections de personnes. 134
B.6 Précision de HMAX en utilisant différentes optimisations. 136
B.7 Utilisation des ressources matérielles de HMAX sur un Artix7-200T. . . . 138

To Ryan and Théo.

Chapter 1

General introduction

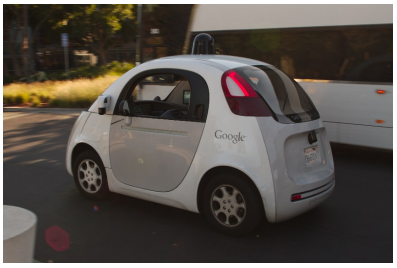
1.1 The need for intelligent systems

Automating tedious or dangerous tasks has been an ongoing challenge for centuries. Many tools have been designed to that end. Among them lies computing machines, allowing to assist human beings in calculations or even performing them. Such machines are everywhere nowadays, in devices that fit into our pockets. However, despite the fact that they are very efficient for mathematical operations that are complicated for our brains, they usually perform poorly at tasks that are easy for us, such as recognizing a landmark on a picture or analysing and understanding a scene.

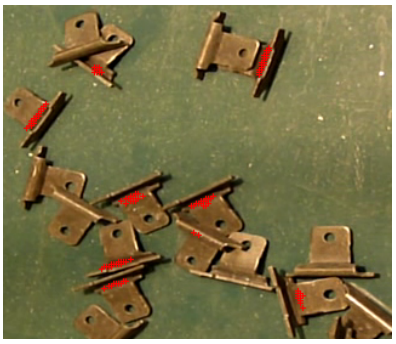
There are many applications for systems that are able to analyze their environments and to make a decision accordingly. In fact, Alan Turing, one of the founder of modern computing, estimated one of the ultimate goal of computing is to build machines that could be said *intelligent* [1]. Perhaps one of the most well known applications of such technology would be for autonomous vehicules, e.g cars that would be able to drive themselves, with little to no help from humans. In order to drive safely, those machines obviously need to retrieve information from different channels, e.g audio of video. Such systems may also be useful for access control for areas that need to be secured, or for quality control on production chains, e.g as was proposed for textile products in [2].

One could think of two ways to achieve a machine of that kind: either engineer how it should process the information, or use methods allowing it to learn it and determine it automatically. Those techniques form a research fields that have been active for decades called *Machine Learning*, which is part of the broader science of *Artificial Intelligence* (AI).

[1] By Michael Shick - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=44405988.



(A) Google's self driving car[1].



(B) Production control.

(C) Security.

(D) Home automation.

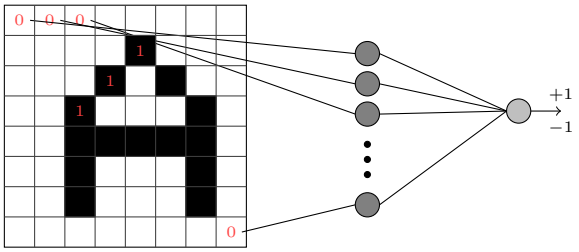
FIGURE 1.1: Application examples.

1.2 Machine Learning

In 1957, the psychologist Frank Rosenblatt proposed the *Perceptron*, one of the first system capable of learning automatically without being explicitly programmed. He proposed a mathematical model, and also built a machine implementing that learning behavior; he tested it with success on a simple letter recognition application. Its principle is very simple: the input image is captured by a retina, producing a small black and white image of the letter – black corresponds to 1, and white to 0. A weighted sum of those pixels is performed, and the *sign* function is applied to the result – for instance, one could state that the system must return 1 when the letter to recognize is an A, and -1 if its a B. If the system returns the wrong value, then the weights are corrected so that the output is correct. A more formal, mathematical description of the Perceptron is provided latter, in Section 2.1.1.1 on page 9. The system is also illustrated in Figure 1.2. Since the Perceptron, many trainable frameworks have been proposed, most of them following a *neuro-inspired* approached like the Perceptron or a statistical approach. They are described in Section 2.1.

Recently, Machine Learning – and AI in general – gained renown from the spectacular research breakthrough and applications initiated by companies such as Facebook, Google, Microsoft, Twitter, *etc.* For instance, Google DeepMind recently developed *AlphaGo*,

(A) Mark I Perceptron[2].



(B) Principle.

FIGURE 1.2: Perceptron applied to pattern recognition. Figure 1.2a shows an hardware implementation, and Figure 1.2b presents the principle: each cell of the retina captures a binary pixel and returns 0 when white, 1 when black. Those pixels are connected to so called input units, and are used to compute a weighted sum. If that sum is positive, then the net returns 1, otherwise it returns -1. Training a Perceptron consists in adjusting its weights. For a more formal and rigorous presentation, see page 9.

a software capable of beating the world champion of Go [3]. Facebook is also using AI to automatically detect, localize and identify faces in pictures [4]. However those applications are meant to be performed on machines with high computational power, and it is beyond question to run such programs on constraint architectures, like those one expect to find on autonomous systems. Indeed, such devices fall into the field of *Embedded Systems* which shall be presented now.

1.3 Embedded systems

Some devices are part of larger systems, in which they perform one task in particular – e.g control the amount of gas that should be injected in the motor of a vehicle. Those so-called *embedded systems* must usually meet high constraints in terms of volume, power consumption, cost, timing and robustness. Indeed, they are often used in autonoumous systems carrying batteries with limited power. In the case of mass produced devices such as phones or cars, it is crucial that their cost is as low as possible. Furthermore, they

[2]By Arvin Calspan Advanced Technology Center; Hecht-Nielsen, R. Neurocomputing (Reading, Mass.: Addison-Wesley, 1990).

are often used in critical systems, where they must process information and deliver the result on time without error – any malfunction of those systems may lead to disastrous consequences, especially in the case of autonomous vehicles or military equipments. All those constraints also mean that embedded systems have very limited computational power.

Many research teams have proposed implementations of embedded intelligent systems, as shown in Section 2.2.2. The work proposed in this thesis falls into that research field. However, as we shall see many of those implementations require high-end hardware, thus leading to potentially high cost devices. The NeuroDSP project[3], in the frame of which this PhD thesis was carried out, aims to provide a device at a lower cost with a low power consumption.

1.4 NeuroDSP: a neuro-inspired integrated circuit

The goal of the research project of which this PhD is part of is to design a chip capable of performing the computation required by the "intelligent" algorithms presented earlier. As suggested in its name, NeuroDSP primarily focuses on the execution of algorithms based on the neural networks theory, among which lie the earlier mentioned Perceptron. As shown in Section 2.1, the main operators needed to support such computations are linear signal processing operators such as convolution, pooling operators and non-linear functions. Most *Digital Signal Processing* (DSP) operators, such as convolution, actually need similar features – hence that device shall also be able to perform DSP operation, for signal preprocessing for instance. As we shall see, all those operations may be, most of the time, performed in parallel, thus leading to a single-instruction-multiple-data (SIMD) architecture, in which the same operations is applied in parallel to a large amount of data. The main advantage of this paradigm is obviously to carry those operations faster, potentially at a lower clock frequency. As the power consumption of a device is largely related to its clock frequency, SIMD may also allow a lower power consumption.

NeuroDSP is composed of 32 so called P-Neuro blocks, each basically consisting of a cluster of 32 *Processing Elements* (PE), thus totalling 1024 PE. A PE may be seen as an *artificial neuron* performing a simple operation on some data. All PEs in a single P-Neuro perform the same operation, along the lines of the aforementioned SIMD paradigm. A NeuroDSP device may then carry out signal processing and decision making operations. Since 1024 neurons may not be enough, they may be multiplexed to emulate larger systems – of course at a cost in terms of computation time. When timing is so

[3] http://goo.gl/Ax6CoF

FIGURE 1.3: NeuroDSP architecture [5]. A NeuroDSP device is composed of 32 clusters, called *P-Neuro*, each constituted of 32 artificial neurons called PE, thus representing a total of 1024 neurons. The PEs may be multiplexed, so that they can perform several instruction sequentially and thus emulate bigger neural networks. When timing is critical, one may instead cascade several NeuroDSP processors and use them as if it was a single device.

critical that multiplexing is not a satisfying option, it is possible to use several NeuroDSP devices in cascade. The device's architecture is illustrated in Figure 1.3.

1.5 Document overview

While NeuroDSP was designed specifically to run signal processing and decision making routines, such algorithms are most of the time too resource consuming to be performed efficiently on that type of device. It is therefore mandatory to optimize them, which is the main goal of the research work presented here.

In Chapter 2, a comprehensive tour of the works related to our research is proposed. After presenting machine learning theoretical background and also algorithms inspired by biological data, the main contribution concerning their implementations are shown. A discussion shall also be proposed, from which arises the problematic that is aimed to be addressed in this document, namely: how may a preprocessing algorithm be optimized given particular face and pedestrian detection applications, and how the data may be efficiently encoded so that few hardware resources may be used?

The first part of that problem is addressed in Chapter 3. While focusing on a preprocessing algorithm called HMAX, the main works in the literature concerning feature selection are recalled. Our contribution to that question is then proposed.

Chapter 4 presents our contribution of the second part of the raised problems, concerning data encoding. After reminding the main research addressing that issue, we show how a preprocessing algorithm may be optimized so that it may process data coded on a few bits only, with few to none performance drop. An implementation on a reconfigurable hardware shall then be proposed.

Finally, Chapter 5 draws final thoughts and conclusions about the work proposed here. The main problems and results are reminded, as well as the limitations. Considered future research are also proposed.

Chapter 2

Related works and problem statement

This chapter proposes an overview of the frameworks used in the pattern recognition field. Both its theoretical backbone and the main implementation techniques shall be presented. It is shown here that one of the key problems of many PR frameworks is their computational cost. Those approaches mainly consists in either using machines with high parallel processing capabilities and high computational power, or on the contrary in optimizing the algorithms so they can be run with less resources. The problematics underlying the work proposed in this thesis, which follows the second paradigm, shall also be stated.

2.1 Theoretical background

In this section, the major theoretical contributions to PR are presented. The principle classification frameworks are first presented to the reader. Then, a description of several descriptors which aim to capture the useful information from the processed images and to get rid of the noise, is proposed.

2.1.1 Classification frameworks

The classification of an unknown data, also called *vector* or *feature vector*, consists in predicting the category it belongs to. Perhaps the simplest classification framework there is is Nearest Neighbor. It consists in storing examples of feature vectors in memory, each associated with the category it belongs to. To classify a unknown feature vector, one

simply uses a distance (e.g Euclidean or Manhattan) to determine the closest example. The classifier then returns the category associated to that selected vector. While really simple, that framework however has many issues. The most obvious is its memory print and its computational cost: the more examples we have, the more expansive that framework is. From a theoretical point of view, that framework is also very sensitive to outliers; any peculiar feature vector, for instance in the case of labelling error, may lead to disastrous classification performance. A way to improve this framework is to take not only *the* closest feature vector, but the K closest, and to make them vote for the category. The retained category is then the one having the most votes [6]. That framework is called *K-Nearest Neighbour* (KNN). While this technique may provide better generalization and reduce the effects due to outliers, it still requires lots of computational resources.

There exist many more other pattern classification frameworks. The most used of those frameworks shall now be described. Neural networks are presented first. A presentation of the Support Vector Machines framework shall follow. Finally, Ensemble Learning methods are presented. This document focuses on feedforward architecture only – non-feedforward architectures, such as Boltzmann Machines [7, 8], Restricted Boltzmann Machines [9, 10] and Hopfield networks [11] shall not be described here. We also focus on *supervised learning* frameworks, as opposed to *unsupervised learning*, such as self-organizing maps [12]. In suppervised learning, each example is manually associated to a category, while in *unsupervised learning* the model "decides" by itself which vector goes to which category.

2.1.1.1 Neural Networks

Artifical Neural Networks (NN) are machine learning frameworks inspired by biolocical neural systems, used both for classification and regression tasks. Neural networks are formed of units called *neurons*, interconnected to each others by *synapses*. Each synapse has a *synaptic weight*, which represents a parameter of the model that shall be tuned during training. During prediction, each neuron performs a sum of its inputs, weighted by the synaptic weights. A non linear function called *activation function* is then applied to the result, thus giving the neuron's *activation* which feeds the neurons connected to the outputs of the considered one. In this thesis, only *feedforward* network shall be considered. In those systems, neurons are organized in successive *layers*, where each unit in a layer gets inputs from units in the previous layer and feeds its activation to units in the next layer. The layer getting the input data is called *input layer*, while the layer from which the network's prediction is read is the *output layer*. Such a framework is represented in Figure 2.1. For a complete overview of the existing neural networks, a good review is given in [13].

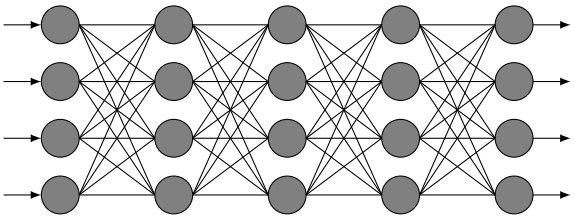


FIGURE 2.1: A feedforward architecture. In each layer, units get their inputs from neurons in the previous layer and feed their outputs to units in the next layer.

Perceptron The perceptron is one of the most fundamental contribution to the Neural Network field, and was introduced by Rosenblatt in 1962 in [14]. It is represented in Figure 2.2. It has only two layers: the input layer and the output layer. A "dummy" unit is added to the input layer, the activation of which is always 1 – the weight w_0 associated to that unit is called *bias*. Those layers are fully connected, meaning each output unit is connected to all input units. Thus, the total input value z of a neuron with N inputs and a bias w_0 is given by:

$$z = w_0 + \sum_{i=1}^{N} w_i x_i \tag{2.1}$$

Or, in an equivalent, more compact matrix notation:

$$z = W^{\mathrm{T}}\mathbf{x} \tag{2.2}$$

with $\mathbf{x} = (1, x_1, x_2, \ldots, x_n)^{\mathrm{T}}$ and $W = (w_0, w_1, w_2, \ldots, w_N)^{\mathrm{T}}$. W is called *weight vector*. In the case where there is more than one output unit, then W becomes a matrix where the i-th column is the weight vector for the i-th output unit. By denoting M the number of output units, z_i the input value of the i-th output unit and $\mathbf{z} = (z_1, z_2, \ldots, z_M)$, one may write:

$$\mathbf{z} = W^{\mathrm{T}}\mathbf{x} \tag{2.3}$$

The output unit's activation function f is as follows:

$$\forall x \in \mathbf{R}, \quad f(x) = \begin{cases} +1 & x > \theta \\ 0 & x \in [-\theta, \theta] \\ -1 & x < \theta \end{cases} \tag{2.4}$$

Where θ represents a *threshold* ($\theta \geq 0$)[1].

To train a Perceptron, it is fed with each feature vector $\mathbf{x}$ in the training set along with the corresponding target category t. Let's consider for now that we only have two different categories: $+1$ and -1. The idea is that, if the network predicts the wrong

[1] In the literature the definition of the activation function may be slightly different, with "$\geq$" signs instead of "$>$" in Equation 2.4 and with $\theta > 0$.

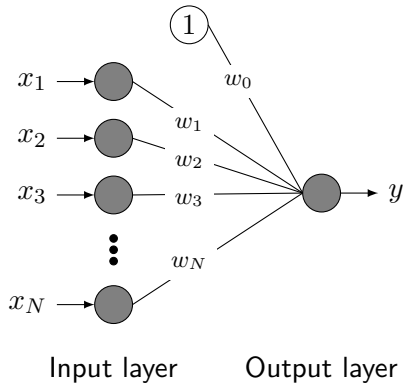


FIGURE 2.2: Perceptron.

category, the difference between the target and the prediction, weighted by a *learning rate* and the input value, is added to the weights and bias. If the prediction is correct, then no modifications is made. The training algorithm is shown in more details for a Perceptron having a single output unit in Algorithm 1. It is easily extensible to systems with several output units; the only major difference is that t is replaced by a target vector $\mathbf{t}$, the components of which may be $+1$ or -1.

$n \leftarrow$ number of input units;
$\eta \leftarrow$ learning rate;
Initialize all weights and bias to 0;
while Stopping condition is false **do**
  forall $(\boldsymbol{x} = (x_1, x_2, \ldots, x_n), t)$ **in** training set **do**
    $y \leftarrow f(w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n)$;
    for $i \leftarrow 1$ **to** n **do**
      $w_i \leftarrow w_i + \eta x_i (t - y)$;
    end
    $w_0 \leftarrow w_0 + \eta (t - y)$;
  end
end

Algorithm 1: Learning rule for a perceptron with one output unit.

If there exists a hyperplan separating the two categories, then the problem is said *linearly separable*. In that case, the perceptron convergence theorem [13, 15–17] states that such a hyperplan shall be found in a finite number of iterations – even if one cannot now that number *a priori*. However, that condition is *required*, meaning the perceptron is not able to solve non-linearly separable problems. Therefore, it is not possible to train a perceptron to perform the XOR operation. This is often referred to as the "XOR problem" in the literature, and was one of the main reasons why neural network had not known great popularity in industrial applications in the past. A way to address this class of problems is to use several layers instead of a single one.

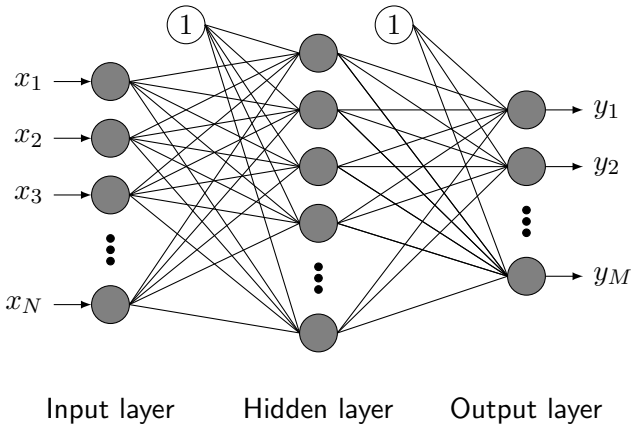


FIGURE 2.3: Multi-layer perceptron with one hidden layer.

Multilayer Perceptron In terms of architecture, the Multilayer Perceptron (MLP) is very similar to the single-layer perceptron, except that it has one or several *hidden layers* between its input and output layers, as shown in Figure 2.3. That architecture allows it to address non linearly separable problems, contrary to the single-layer Perceptron. As required by its training algorithm, its neurons' activation function must be defined and derivable on $\mathbf{R}$, and staked in $[-1, +1]$. Therefore, f may be the hyperbolic tangent:

$$\forall x \in \mathbf{R} \quad f(x) = \tanh(x) \tag{2.5}$$

or the very similar bipolar sigmoid:

$$\forall x \in \mathbf{R} \quad f(x) = \frac{2}{1 + e^{-x}} - 1 \tag{2.6}$$

Those functions' curves are represented in Figure 2.4. Its training algorithm is somewhat more complicated, and follows the *Stochastic Gradient Descent* approach. Let E be the *cost function* measuring the error between the expected result and the network's prediction. The goal is to minimize E, the shape of which is unknown. The principle of the algorithm achieving that is called *back-propagation of error* [18, 19].

RBF Radial Basis Function networks were proposed initially by Broomhead and Lowe [20, 21] and fall in the *kernel methods* family. They consist in three layers: an input layer similar to the Perceptron's, a hidden layer containing kernels and an output layer. Here, a kernel i is a radial basis function f_i (hence the name of the network) that measures the proximity of the input pattern $\mathbf{x}$ with a learnt pattern $\mathbf{p}_i$ called *center*, according to a *radius* β_i. It typically has the following form:

$$f_i(x) = \exp\left(-\frac{||\mathbf{x} - \mathbf{p}_i||}{\beta_i}\right) \tag{2.7}$$

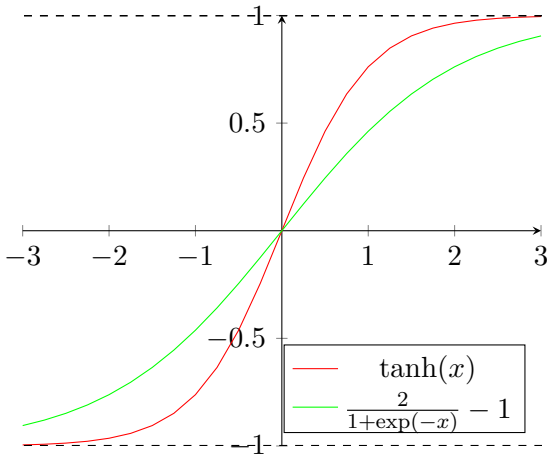


FIGURE 2.4: MLP activation functions.

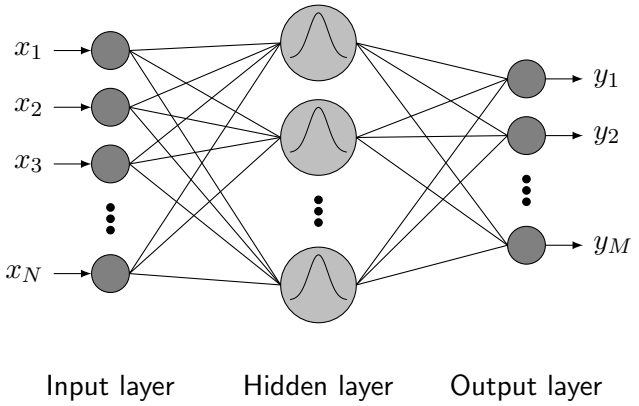


FIGURE 2.5: RBF neural network.

The output layer is similar to a Perceptron: the hidden and output units are fully connected by synapses having synaptic weights, which are determine during the training stage. The network is illustrated in Figure 2.5.

To determine the kernels parameters, one may adopt different strategies. Centers may be directly drawn from the training set, and radius may be arbitrarily chosen – however such empirical solution leads to poor results. A more efficient way is to use a clustering algorithm that gathers the centers into clusters, the center of which shall represent an example center while the corresponding radius is evaluated w.r.t the proximity with other kernels. Such an algorithm is presented in Appendix A. The computational power and the memory required by this network grows linearly with the number of kernels. While the training method presented in Appendix A tend to reduce the number of kernels, it still may be quite important. There exists sparse kernel machines, that work in a similar way than RBF networks but are designed to use as few kernels as possible, like the Support Vector Machines described in Section 2.1.1.2.

Spiking Neural Network All the models presented above treat the information at the level of the neurons activation. Spiking neural networks intend to describe the behaviour of the neurons at a lower level. That model was first introduced by Hodgkin *et al* [22], who proposed a description of the propagation of the action potentials between biological neurons. There exists different variations of the spiking models, but the most used nowadays is probably the "integrate and fire", where the neurons' inputs are accumulated over time. When the total reaches a threshold, the neuron is committed. Thus, the information sent by a neuron is not carried by a numerical value, but rather by the spikes order and the duration between two spikes. It is still an active research subject, with many applications in computer vision – Masquelier and Thorpe proposed the "spike timing dependent plasticity" (STDP) algorithm, which allows unsupervised learning of visual features [23].

2.1.1.2 Support Vector Machine

Support Vector Machines (SVM) are linear classifiers. Much like any other such model, they aim to find the parameters of the hyperplan separating best a category with the others. It may be used in conjunction with kernel functions, in the same way as presented in the description of RBF neural networks in Section 2.1.1.1, page 11. SVM training's algorithm aims to determine the vectors that are the nearest to those of the other categories [24]. The selected vectors are called *support vectors*. After selecting them, the decision boundary's parameters are optimized so that it is as far as possible to all support vectors. Typically, a quasi-Newton optimization process could be choosen to that end; however its description lies beyond the scope of this document. Figure 2.6 shows an example of their determination as well as the resulting decision boundary.

2.1.1.3 Ensemble learning

The rational behind Ensemble Learning frameworks is that instead of having one classifier, it may be more efficient to use several ones [25–28]. Those classifiers are called *weak classifiers*, and the final decision results from their predictions. Their exists several paradigms, among which *Boosting* [29] in particular.

Boosting algorithm are known for their computational efficiency during prediction. A good example is their use in Viola and Jone's famous face detection algorithm [30]. The speed of the algorithm comes partly from the fact that the classifier is composed of a cascade of weak classifiers, in which all regions of the image that are clearly not faces are discarded by the top-level classifier. If the data goes through it, then it is "probably a face", and is processed by the second classifier, which either discard of accept it, and

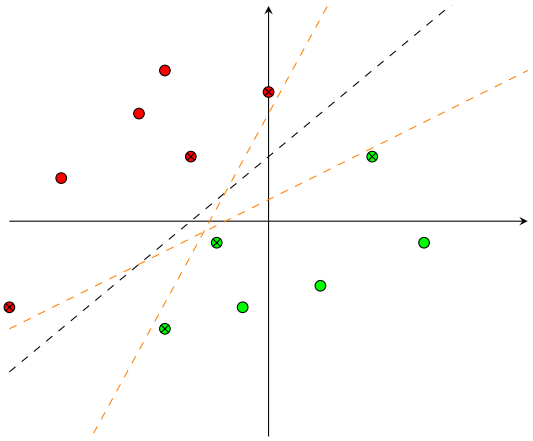


FIGURE 2.6: Support vectors determination. Green dots belong to a class, and red ones to the others. Dots marked with a × sign represent the selected support vectors. The unmarked dots have no influence over the determination of the decision boundary's parameters. The black dashed line represents the determined decision boundary, and the orange lines possible decision boundaries that would not be optimal.

so on. This allows to rapidly eliminate irrelevant data and the noise. Boosting is also known to be slightly more efficient than SVM for multiclass classification tasks with HMAX [31], which is described in Section 2.1.2.2.

2.1.2 Feature extraction frameworks

2.1.2.1 Signal processing approach

Classical approaches More than ten years ago, Lowe proposed a major contribution in computer vision with his Scale Invariant Feature Transform (SIFT) descriptor [32], which became quickly very popular due to its efficiency. Its primary aim was to provide, as suggested by its name, features that are invariant to the scale and to some extent to the orientation and small changes in viewpoint. It consists in matching features from the unknown image to a set of learnt features at different locations and scales, followed by a Hough transform that gathers the matched points in the image into clusters, which represent detected objects. The matching is operated by a fast nearest-neighbour algorithm, that indicates for a given feature the closest learnt feature. However, doing so at every locations and scale would be very inefficient, as most of the image probably does not contain much information. In order to find locations which are the most likely to hold information, a Difference of Gaussian (DoG) filter bank is applied to the image. Each DoG filter behaves as a band-pass filter, selecting edges at a specific spatial frequency and allowing to find features at a specific scale. Extrema are then evaluated across all those scales in the whole image, and constitute a set of keypoints at which the aforementioned matching operations are performed. As for rotation invariance, it

is brought by the computation of gradients that are local to each keypoint. Before performing the actual matching, the data at a given keypoint is transformed according to those gradients so that any variability caused by the orientation is removed.

Bay *et al.* proposed in [33] a descriptor aiming to reproduce the result of the state of the art algorithm, but much faster to compute. They called their contribution SURF, for Speeded-Up Robust Features. It provides properties similar to SIFT (scale and rotation invariance), with a speed-up of 2.93X on a feature extraction task, where both frameworks were tuned to extract the same number of keypoints. Like SIFT, SURF consists in a detector that takes care of finding keypoints in the image, cascaded with a descriptor that computes features at those keypoints. The keypoints are evaluated using a simple approximation of the Hessian matrix, which can be efficiently computed thanks to the integral image representation, i.e an image where each pixels contains the sum of all the original image's pixels located left and up to it [30]. Descriptors are then computed locally using Haar wavelet, which can also be computed with the integral image [30]. [34, 35]

Another popular framework for feature extraction is Histograms of Oriented Gradients (HOG) [36]. It may be used in many object detection applications, though it was primarly designed for the detection of human beings. It consists in computing the gradients at each pixel, and make each of those gradients vote for a particular bin of a local orientation histogram. The weight with which each gradient votes is a linear function of its norm and of the difference between its orientation and the orientation of the closest bins' centers. Those gradients are then normalized over overlapping spatial blocks, and the result forms the feature vector. The classifier used here is typically a linear SVM, presented in Section 2.1.1.
Like many feature extraction frameworks, there exists some variations of the HOG feature descriptor. Dalal and Triggs present two of them in [36]: R-HOG and C-HOG, respectively standing for "Rectangular HOG" and "Circular HOG". The difference with the HOG lies in the shape of the overlapping spatial blocks used for the gradient normalization. R-HOG is somewhat close to presented earlier SIFT, except that computations are performed at all locations, thus providing a dense feature vector. C-HOG is somewhat trickier to implement due to the particular shape it induces, and shall not be presented here. All three frameworks provide similar recognition performances, which were the state of the art at that time.

There are many other descriptors for images, like FAST [34, 35], and we shall not describe them in detail here as it lies beyond the scope of this document. However it is worth detailing another type of frameworks based on so-called *wavelets*, which allow to retreive

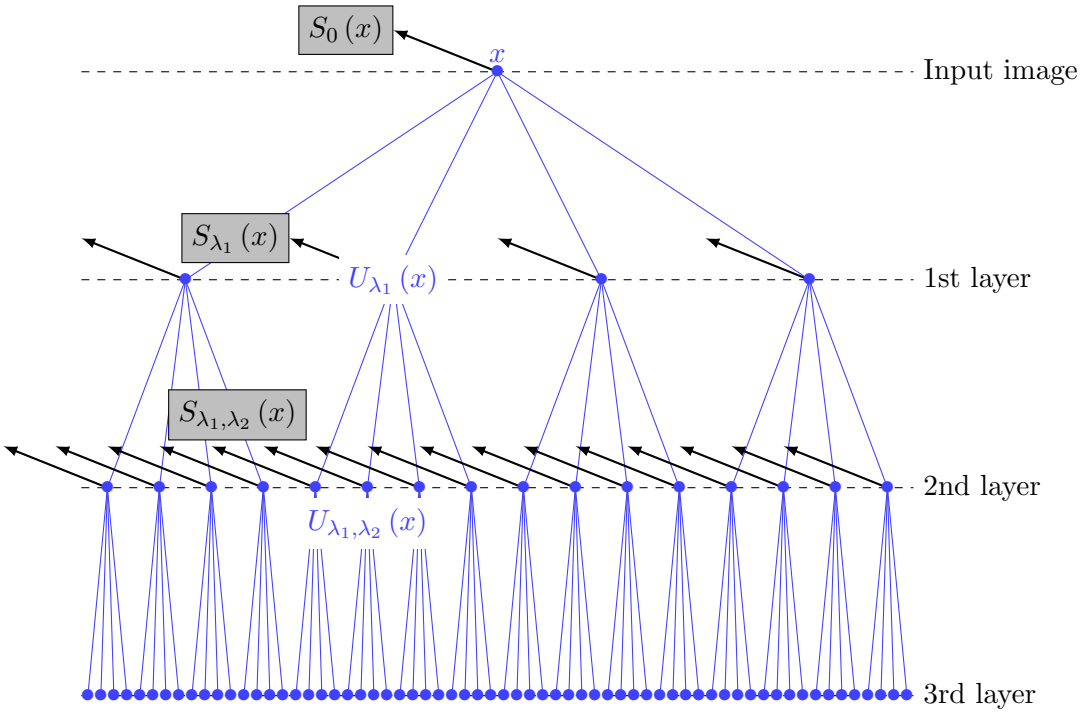


FIGURE 2.7: Invariant scattering convolution network [38]. Each layer applies a wavelet decomposition U_λ to its inputs, and feed the next layer with the filtered images $U_\lambda(x)$. At each layer, a low-pass filter is applied to the filtered images and the results are sub-sampled. The resulting so-called "scattering coefficients" $S_\lambda(x)$ are kept to form the feature vector.

frequency information while keeping local information – which is not possible with the classical Fourier transform.

Wavelets Wavelets have known a great success in many signal processing applications, such as signal compression or pattern recognition, including for images. They are linear operators decomposing locally a signal on a frequency basis. A wavelet decomposition consists in applying a "basis" linear filter, called *mother wavelet*, on the signal. It is then dilated in order to extract features of different sizes and, in the case of images, rotated so that it responds to different orientations. An excellent and comprehensive guide to the theory and practice of wavelets is given in [37].

Wavelets are used as the core operators of the Scattering Transform frameworks. Among them lie the Invariant Scattering Convolution Networks (ISCN), introduced by Bruna and Mallat [38]. They follow a feedforward, multistage structure, along the lines of ConvNet described in Section 2.1.2.3, though contrary to ConvNet its parameters are fixed, not learnt. They alternate wavelet decompositions with low-pass filters and sub-sampling – the function of which is to provide invariance in order to raise classification performances. Each stage computes a wavelet decomposition of the images produced

at the previous stage, and feed the resulting filtered images to the next stage. At each stage the network also outputs a low-pass filtered and sub-sampled version of those decompositions – the final feature vector is the concatenation of those output features. Figure 2.7 sums up the data flow of this framework. It should be noted that in practice, not all wavelet are applied at each stage to all images: indeed it is shown in [38] that some of those wavelet cascades do not carry information, and thus their computation may be avoided, which allows to reduce the algorithmic complexity. Variations of the ISCN with invariance to rotation are also presented in [39, 40], which may be used for texture [39] or objects [40] classification.

2.1.2.2 A biological approach: HMAX

Some frameworks are said to be *biologically plausible*. In such case, their main aim is not so much to provide a framework as efficient as possible in terms of recognition rates or computation speed, but rather to propose a model of a biological system. One of the most famous of such frameworks is HMAX, which also happens to provide decent recognition performances. The biological background was proposed by Riesenhuber and Poggio in [41], on the base of the groundbreaking work of Hubel and Wiesel [42]. Its usability for actual object recognition scenarios was stated by Serre *et al.* 8 years later in [31]. It is a model of the ventral visual system in the cortex of the primates, accounting for the first 100 to 200 ms of processing of visual stimuli. As its name suggests – HMAX stands for "Hierarchical Max" – that model is built in a hierarchical manner. Four successive stages, namely S1, C1, S2 and C2 process the visual data in a feedforward way. The S1 and S2 layers are constituted of *simple cells*, performing linear operations or proximity evaluations, while the C1 and C2 contain *complex cells* that provide some degrees of invariance. Figure 2.8 sums up the structure of this processing chain. Let's now describe each stage in detail.

The S1 stage consists in a Gabor filter bank. Gabor filters – which are here two dimensional, as we process images – are linear filters responding to patterns of a given spatial frequency and orientation. They are a particular form of the wavelets described in Section 2.1.2.1. A Gabor filter is described as follows:

$$G(x,y) = \exp\left(-\frac{x_0^2+\gamma^2 y_0^2}{2\sigma^2}\right) \times \cos\left(\frac{2\pi}{\lambda}x_0\right) \tag{2.8}$$

$$x_0 = x\cos\theta + y\sin\theta \quad \text{and} \quad y_0 = -x\sin\theta + y\cos\theta \tag{2.9}$$

where γ is the filter's aspect ratio, θ its orientation, σ the Gaussian effective width and λ the cosine wavelength. The filter bank has several filters, each having a specific wavelength, effective width, size and orientation. The wavelength, effective width and

size define the filter's *scale*. There are 16 different scales and four different orientations, thus totaling 64 filters. During the S1 stage, each filter is applied independently on the input image and the filtered images are fed to the next layer.
The C1 stage gives a first level of location invariance of the features extracted in S1. It does so with maximum pooling operators: each C1 unit pools over several neighboring S1 units with a 50% overlap and feed the S2 layer with the maximum value. The number of S1 units a C1 unit pools over depends on the scale of the considered S1 units. Furthermore, each C1 unit pools across two consecutive scales, with no overlap. This leads to a number of images divided by two, thus only 32 images are fed to the following layer. The parameters of the S1 and C1 layers are presented in Table 2.8.
The S2 stage aims to compare the input features to a dictionary of learnt features. There are different ways to build up that dictionary. In [31] it is proposed to simply crop patches of different sizes in images in C1 space at random position and scales. During feedforward, patches are cropped from images in C1 space at *all* locations and scales, and are compared to each learnt feature. The comparison operator is a radial basis function, defined as follows:

$$\forall i \in \{1, 2, \ldots, N\} \quad r_i\left(\mathbf{X}\right) = \exp(-\beta \left\|\mathbf{X} - \mathbf{P}_i\right\|) \tag{2.10}$$

where $\mathbf{X}$ is the input patch from the previous layer, $\mathbf{P}_i$ the i-th learnt patch in the dictionary and $\beta > 0$ is a tuning parameter. Therefore, the closer the input patch is to the S2 unit learnt patch, the stronger the S2 unit fires.
Finally, a complete invariance to locations and scales of the features in C1 space is reached in the C2 stage. Each C2 unit pools over all S2 unit sharing the same learnt pattern, and simply keeps the maximum value. Those values are then serialized in order to form the feature vector. The descriptor HMAX provides is well suited to detect the presence of an object in cluttered images, though the complete invariance to location and scales brought by C2 removes information related to its location. This issue is addressed in [43] – however that model lies beyond the scope of this thesis and shall not be discussed here. Different variations of this model have been proposed; among them, of particular interest are the sparse version proposed by Mutch and Lowe [44, 45] and Yu and Slotine's wavelet-based, speeded-up version [46].

2.1.2.3 Convolutional Neural Network

Convolutional Neural Networks, also known as ConvNet or CNN, were principally introduced by Yann LeCun [47, 48, 48, 49]. It is a hierarchical architecture largelly inspired, like HMAX (see Section 2.1.2.2), by the structure of the visual cortex in mamals. It consists in a succession of *convolution* and *subsampling* layers that respectively behave

C1 layer			S1 Layer		
Scale band	Spatial pooling grid ($N_k \times N_k$)	Overlap Δ_k	filter size k	Gabor σ	Gabor λ
Band 1	8×8	4	7×7 9×9	2.8 3.6	3.5 4.6
Band 2	10×10	5	11×11 13×13	4.5 5.4	5.6 6.8
Band 3	12×12	6	15×15 17×17	6.3 7.3	7.9 9.1
Band 4	14×14	7	19×19 21×21	8.2 9.2	10.3 11.5
Band 5	16×16	8	23×23 25×25	10.2 11.3	12.7 14.1
Band 6	18×18	9	27×27 29×29	12.3 13.4	15.4 16.8
Band 7	20×20	10	31×31 33×33	14.6 15.8	18.2 19.7
Band 8	22×22	11	35×35 37×37	17.0 18.2	21.2 22.8

TABLE 2.1: Paramaters for HMAX S1 and C1 layers [31]. Concerning the Gabor filters in S1, σ represents the spread of their Gaussian envelopes and λ the wavelength of their underlying cosine functions.

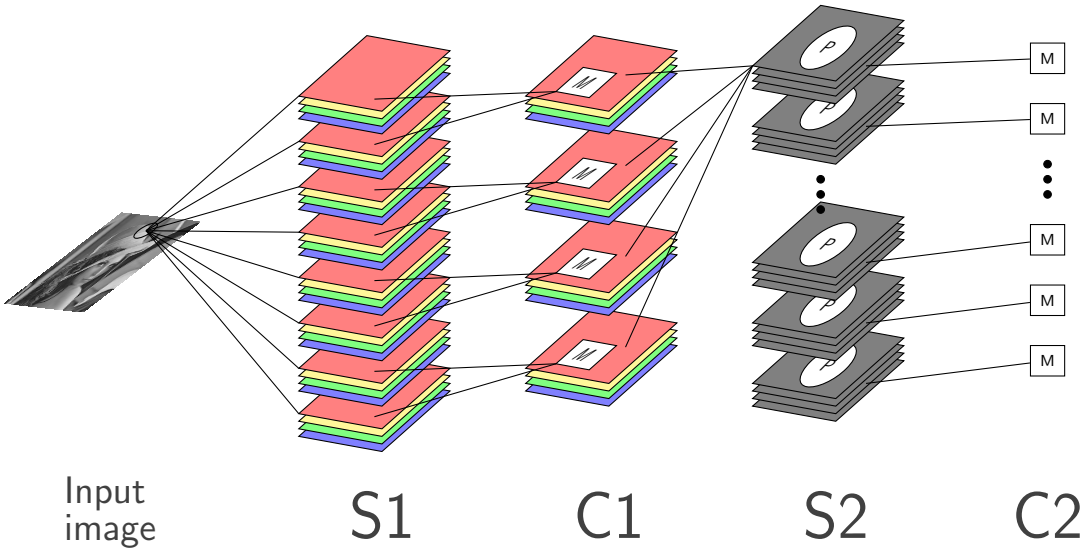


FIGURE 2.8: HMAX [31]. Its feedforward architecture is composed of 4 layers: S1, C1, S2 and C2. S1 applies a Gabor filter bank to the input image, each filter having a preferred scale and orientation. The different colors represent the different orientations, and each set of 4 image represents a scale. There are 16 scales and 4 orientations – only 8 scales are represented here for readability reasons. The C1 layer units pool across neighboring S1 units of successive scales and keep the maximum response. The S2 layer matches patches produces by C1 with prelearnt patches, and the C2 layer keeps, for each prelearnt patch in S2, its maximum response.

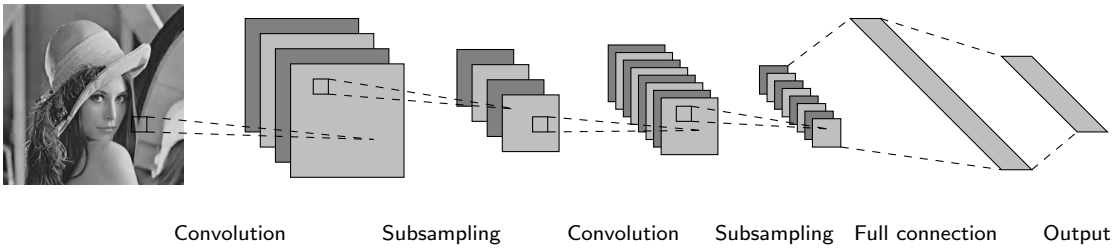


FIGURE 2.9: Convolutional neural network [48].

like the S1 and C1 layers of HMAX, followed by a fully connected layer similar to a MLP. However, the parameters of the convolution kernels are not predefined, but rather learnt at the same time as the weights in the final classifier. Thus, the feature extraction and classification models are both tuned simultaneously, using an extension of the back-propagation algorithm.. An example of this model is presented in Figure 2.9. That framework became very popular since the industry demonstrated its efficiency, and is today actively used by big companies such as Facebook, Google, Twitter, Amazon and Microsoft. A particular implementation of that framework, tuned to perform best at face recognition tasks, was proposed by Garcia *et al* [50]. However, the large amount of parameters to be optimized by the training algorithm requires a huge amount of data in order to avoid overfitting, lots of computational power and lots of time – still, pretrained models are provided by the community, making that problem avoidable.

2.2 Frameworks implementations

2.2.1 Software implementations

There exists many implementation of the descriptors and classifier described in Section 2.1. Some of them are available in general purpose software packages, like the widespread Scikit-learn python package [51]. SVM also have a high performance dedicated library with LIBSVM [52]. Other frameworks, more dedicated to neural networks – and particularly deep learning – are accelerated on GPUs, like Theano [53, 54], Caffe [55], Torch [56], cuDNN [57] and the recently released TensorFlow [58]. There also exist frameworks more oriented towards neuroscience, such as PyNN [59] and NEST [60]. The Parallel Neural Circuit Simulator (PCSIM) allows to handle large-scale models composed of several networks that may use different neural models, and is able to handle several millions of neurons and synapses [61]. As for spiking neural networks, the BRIAN framework [62, 63] provides an easy to use simulation environment.

2.2.1.1 Workstations

Mutch *et al.* proposed in 2010 the Cortical Network Simulator framework [64], aiming to automatically compile algorithms following a cortical architecture, such as HMAX. The latter also has an optimized GPU implementation presented in [65], achieving real time image classification on a NVIDIA Geforce 9400M GPU. A new neural network model similar to ConvNet, called Locally-connected Neural Pyramid (LNCP) was proposed by Uetz and Behnke along with its implementation on GPU [66], using the CUDA framwork. This framework was especially designed for large-scale object recognition. The authors claim a very low testing error rate of 0.76 % on MNIST, a popular hand-written digit dataset initially provided by Burges *et al* [67], and 2.87 % on the general purpose NORB dataset [68].

2.2.1.2 Embedded systems

Optimizations for software implementations, both on CPU and GPU, for the SIFT and SURF frameworks have also been proposed [69]. It has also been shown that wavelets are very efficient to compute, even on low hardware resources [70], which make them a reasonable choice for feature extraction on embedded systems. Furthermore, an embedded version of the SpiNNaker board described in Section 2.2.2 for autonomous robots, programmable using with the C language or languages designed for neural networks programing is presented in [71].

2.2.2 Hardware implementations

As shown in Section 2.2.1, GPUs are very efficient platforms for the implementation of classification and feature extraction frameworks, particularly for neuromorphic algorithms, due to their highly parallel architecture. Field Programmable Gate Arrays (FPGA) are another family of massively parallel platforms, and as such are also good candidates for efficient implementations. They are reconfigurable hardware devices, in which the user implement algorithms at a hardware level. Therefore, they provide a much finer control than the GPU: one implements indeed the communication protocols, the data coding, how computations are performed, *etc.* – though they utilization is also more complicated. FPGAs are configured using *hardware description languages*, like VHDL or Verilog.

Going further down in the abstraction levels, there also exists fully analogical neural network implementations that use a component called *memristor* [72–81]. The resistance of such components can be controlled by the electric charge that goes through it. That

resistance value is analogous to a synaptic weight. As it is still at the fundamental research level, analogical neural network shall not be studied here.

2.2.2.1 Neural networks

The literature concerning hardware implementations of neural networks is substantial. A very interesting and complete survey was published in 2010 by Misra *et al* [82]. Feedforward neural network are particularly well suited for hardware implementations, since the layers are, by definition, computed sequentially. It implies that the data goes through each layers successively, and that while the layer i processes the image k, the image $k+1$ is processed by the layer $i-1$. Another strategy is, on the contrary, to implement a single layer on the device, and to use layer multiplexing to sequentially load and apply each layer to the data, thus saving lots of hardware resources to the expense of a higher processing time [83]. However, it has been demonstrated that neural network that are not feedforward may also be successfuly implemented on hardware [84, 85]. There also exist hardware implementations of general purpose bio-inspired frameworks, such as Perplexus, which proposes among other the capability for hardware devices to self-evolve, featuring dynamic routing and automatic reconfiguration [86], particularly suited for large-scale biological system emulation. Architecture of adaptive size have also been proposed, that allow to dynamically scale itself when needed [87].
While the mentioned works intend to be general purpose frameworks with no particular applications in mind, some contributions also propose implementations for very specific purposes, such as the widespread face detection and identification task [88], or more peculiar application such as gas sensing [89] or classification of data acquired from magnetic probes [90].

Some frameworks received special considerations from the community in those attempts. After presenting the works related to HMAX, the next paragraphs shall present the numerous – and promising – approaches for ConvNet implementations. The many contributions that concern the Spiking Neural Networks are presented afterwards.

HMAX Many contributions about hardware architectures for HMAX have been proposed by Al Maashri and his colleagues [91–96]. Considering that in HMAX, the most resource consuming stage is, by far, the S2 layer [92], a particular effort was made in [92] to propose a suitable hardware accelerator for that part. In that paper, Al Maashri *et al.* proposed a stream-based correlation, where input data is streamed to several pattern matching engines performing the required correlation operations in parallel. The whole model, including the other layers, was implemented on a single-FPGA and a multi-FPGA platforms that respectively provide 23× and 89× speedup, compared with

a CPU implementation running on a system having a quad-core 3.2 GHz Xeon processor and 24 GB memory. The single-FPGA platform uses a Virtex-6 FX-130T, and the multi-FPGA one embeds four Virtex-5 SX-240T, all of which are high-end devices. Those systems did not have any drop in accuracy compared to the CPU implementation.

A complete framework allowing to map neuromorphic algorithms to multi-FPGA systems is presented by Park*et al.* in [91]. The chosen hardware platform is called Vortex [97], which was designed to implement and map hardware accelerators for stream-based applications. One of the biggest challenge for such systems is the inter-device communication, which is addressed in that work with the design of specific network interfaces. It also proposes tools allowing to achieve the mapping in a standardized way, with the help of a specially-designed tool called *Cerebrum*. As a proof of concept, a complete image processing pipeline was implemented, that cascades a preprocessing stage, a visual saliency [2] determination and an object recognition module using HMAX. That pipeline was also implemented on CPU in C/C++ and on GPU with CUDA for comparison. The gain provided by the system is a speedup of 7.2× compared to the CPU implementation and 1.1× compared to the GPU implementation. As for the power efficiency, the gain is 12.1× compared to the CPU implementation and 2.3× compared to the GPU implementation.

Kestur *et al* proposed with their CoVER system [98] a multi-FPGA based implementation of visual attention and classification algorithms – the latter being operated by HMAX – that aims to process high resolution images nearly in real time. It has a pre-processing stage, followed by either an image classification or a saliency detection algorithm, or both, depending on the chosen configuration. Each process uses a hardware accelerator running on an FPGA device. The architecture was implemented on a DNV6F6-PCIe prototyping board, which embeds six high-end Virtex6-SX475T FPGAs: one of them is used for image preprocessing and routing data, another one to compute HMAX's S1 and C1 feature maps, two perform the computations of HMAX's S2 and C2 features, and the remaining two are used both as repeaters and to compute the saliency maps.

To our knowledge, the most recent hardware architecture for HMAX was proposed in 2013 by Orchard *et al* [99]. It was successfuly implemented on a Virtex 6 ML605 board, which carries a XC6VLX240T FPGA. The implementation is almost identical to the original HMAX described in [31], and is able to process 190 images per second with less than 1% loss in recognition rate compared with standard software implementations, for both binary and multiclass objet recognition tasks. One of the major innovation of

[2] A *saliency* is a region that is likely to contain information in an image. Saliencies are typically determined with edge detection and the frequency of occurrences of a pattern in the image – the less frequent, the more unusual and thus the more salient that pattern shall be.

this contribution is the use of separable filters for the S1 layer: it was indeed shown that all filters used in HMAX, at least the original version presented in [31], may be expressed as separable filters or as a linear combinations of separable filters – this allows to considerably reduce the utilization of FPGA resources.

ConvNet In 2009, Farrugia *et al* proposed an implementation of a particular version of ConvNet, called "Convolutional Face Finder" (CFF) which is described in Section 3.1.1.2, on FPGA [100]. The architecture is based on an interconnection of processing elements and a FIFO module, following a ring topology. Each PE performs the CFF algorithm on a small chunk of the image, and the ring topology allows to overlap between several regions of the image. The authors tested two implementations: one with 4 PEs on a Virtex 4 SX 35 FPGA that processes 29 grayscale QVGA images[3], and the other one with 25 PEs on a Virtex 5 LX 330 FPGA that processes 127 grayscale QVGA images or 35 grayscale VGA images. All implementations have a clock frequency of 80 MHz. The main advantage of this model is that it is easily scalable: to process bigger images while maintaining the same frame rate, or to raise the frame rate while processing images of the same size, one would just need to add PEs to the network.

In a similar work, Farabet *et al* showed that ConvNets could be implemented on low end FPGAs [101, 102]. They implemented a network constituted of three convolution layers and two pooling layers, processing 42×42 input images, on a board carrying a low-end Spartan-3A DSP 3400 FPGA coupled to a DDR-SDRAM module – though the model could not run at full speed due to bandwidth problem with the communication from and to the memory modules. They also proposed the same implementation on a board carrying a high-end Virtex-4 SX35 FPGA coupled to two QDR-SRAM modules with a higher bandwidth, allowing the system to work at full speed. Both implementations were tested on a face detection application. As the training stage of ConvNets needs a large computational power (see Section 2.1.2.3), it was performed on a usual machine using 30,000 images. At runtime, the high-end system was able to process 10 grayscale VGA images per second while consuming 15 W in peak.

The contribution of Farabet *et al* mentionned above led to a framework called *neuFlow* [103], released in 2011. It was designed to be a generic detection, classification and localization system for computer vision applications. It may be configured thanks to a compiler specifically created called *luaFlow*, allowing to translate a network's high-level graph representation into byte code readable by neuFlow. The framework relies on a dataflow architecture described earlier in [104]; it composed of a matrix of so-called Processing Tiles (PT) – each having a bank of operators and a MUX for communication

[3] With a resolution of 320×240

with other PTs – and a Directory Memory Access (DMA) module for shared memory management. As an example, a street-scene analysis system, able to perform segmentation of street-scene images and assign each pixel a label depending on the nature of the object it belongs to, was implemented with neuFlow on a Virtex 6 ML605 FPGA. The implementation had a 200 MHz clock frequency and consumed 10 W, and was able to fully process 12 images per second. It should be noted that a System on Chip (SoC) version of neuFlow was released in 2012 [105]. It can perform up to 160 GOPS[4] with a power consumption of 512mW.

A low power coprocessor designed for ConvNets was also proposed by Gokhale *et al* in [106], and is called nn-X which stands for Neural Network Next. It was developped by a company called TeraDeep, founded by Yann LeCun among others. It has three main components: a host processor composed of two ARM Cortex-A9 CPUs, the coprocessor itself that carries out the computations needed by the neural network, and an external memory. The coprocessor consists in an array of PEs (called *collections* in the paper), a memory router for communication and a bus for configuration. A collection is itself composed of convolution modules, a pooling module, a router for communication with the shared memory and other collections, and a programmable non-linear function module. The system was run on a ZC706 prototyping board, allowing the instanciation of eight collections. On that board it runs at a clock frequency of 142 MHz, and measured power consumption for the processors, FPGA and memory is 4W.

Convolution are the most resource consuming operations in ConvNets, and as such many of the above work propose optimizations for them. A particular effort towards that goal was provided by Conti *et al* with their HWCE (Hardware Convolution Engine) system. That engine is composed of three submodules: a *wrapper* that takes care of communications with other modules, a *weight loader* that manages the convolution kernel's coefficients and the *convolution engine* itself, that performs the actual computation. In order to perform the convolution operations in streams, the convolution kernel stores a stripe of the image and perform convolutions as soon there are enough data, so that for a $K \times K$ convolution kernel the system needs to store $K - 1$ lines. Thus, the system can output one pixel per clock cycle. That engine reached the to date state-of-the-art in terms of energy efficiency, wih 2.76 GOPS/mW.

To our knowledge, the most recent effort concerning the implementation of ConvNets on hardware lies in the Origami project [107]. The contributors claim that their integrated circuit is low-power enough to be embeddable, while handling network that only workstation with GPU could handle before. To achieve this, the pixel stream is first, if necessary, cropped to a Region Of Interest (ROI) with a dedicated module. A

[4] Giga Operations Per Seconds

filter bank is then run on that ROI. Each filter consists in the combination of chanels, each performing multiplication-accumulation[5] (MAC) operations on the data they get. Each channel then sums the final results individually, and output the pixel values in the stream. That system achieves a high throughput of 203 GOPS when running at 700 MHz, and consumes 744 mW.

Spiking Neural Networks Due to the potentially low computational resources they need, SNN also have their share of hardware implementation attempts. Perhaps the most well-known is the Spiking Neural Network architecture (SpiNNaker) Project [108]. It may be described as a massively parallel machine, capable of simulating neuromorphic systems in real time – i.e it respects biologically plausible timings. It it basically a matrix of interconnected processors (up to 2500 in the largest implementation), splitted in several *nodes* of 18 processors. Each processor simulates 1000 neurons. The main advantage in using spikes is that the information is carried by the firing timing, as explained in Section 2.1.1.1, page 12 – thus each neuron needs to send only small packets to the other neurons. However, the huge amount of those packets and of potential destination makes it challenging to route them efficiently. In order to guarantee that each emitted packet arrives on time at the right destination, the packet itself only contains the identifier of the emitting neuron. Then, the router sends it to the appropriate processors according to that identifier, which would depend on the network's topology and more precisely to which neurons the emitting neuron is connected to.

IBM and the EPFL (*École Polytechnique Fédérale de Lausanne*) collaborated to start a large and (very) ambitious research program: the Blue Brain project, which aims to use an IBM Blue Gene supercomputer to simulate mammalian brains, first of little animals like rodents, and eventually the human brain [109]. However it is highly criticized by the scientific community, mostly for its cost, the lack of realism in the choice of its goals and the contributions it led to [110]. While still ongoing, that project led to the creation of SyNAPSE, meaning *System of Neuromporphic Adaptive Plastic Scalable Electronics*. Since the Blue Brain project needed a supercomputer, the aim of SyNAPSE is to design a somewhate more constrained system. In the frame of that project, the TrueNorth chip [111] was released in 2014. It consists in 4,096 parallel interconnected cores implementing one million spiking neurons. IBM claim that that chip consumes 65 mW for a multi object detection and classification task at 30 FPS, taking as inputs 240×400 images with 3 color channels [112].

Finally, Krichmar *et al.* proposed several case studies for large scale spiking neural networks [113], run in a simulation environment. The authors backed the propositions

[5] A multiplication between an input data and a coefficients, the result of which being added to a data computed before by another MAC operation.

that neural networks may be useful for both engineering and modeling purposes, and supported the fact that the spiking neural networks are particularly well suited with the use of Addressable Event Representation communication scheme, which consists in transmitting only the information about particular events instead of the full information, which is particularly useful to reduce the required bandwidth and computations. However, that strategy lies beyond the scope of this document.

2.2.2.2 Other frameworks implementations

There exists many academic works that are yet to be mentioned, for both classifiers and descriptors. As for classifiers, Kim *et al* proposed a bio-inspired processor for real time object detection, achieving high throughput (201.4 GOPS) while consuming 496 mW. Other frameworks for pattern recognition systems that are not biologically inspired have been proposed. For instance, Hussain *et al* proposed an efficient implementation of the simple KNN algorithm [114], and an implementation of the almost-equally-simple Naive Bayes[6] framework is proposed in [115]. Anguita *et al* proposed a framework allowing to generate user-defined FPGA cores for SVMs [116]. An implementation for Gaussian Mixture Models, which from a computational point of view are somewhat close to RBF nets and as such may require lots of memory and hardware resources, have also been presented [117]. Concerning feature extraction, the popular SIFT descriptor have been implemented on FPGA devices with success [118, 119], as well as SURF [120].

Some companies also proposed their own neural netwok implementations, long before the arrival of ConvNet, HMAX and other hierarchical networks. Intel proposed an analogical neural processor called ETANN in 1989 [121]. While harder to implement and not as flexible as their digital counterparts, analogical devices are much faster. That processors embeds 64 PEs that act as as many neurons and $10,240$ connections. The device was parameterizable by the user using a software called *BrainMaker*. A digital neural architecture was presented by Phillips for the first time in 1992, and was called L-neuro [122, 123]. It was designed with modularity as a primarly concern in mind, and thus is easily interconnected with other modules which makes it scalable. In its latter version, that system was composed of 12 DSP processors, achieving 2 GOP/s with a 1.5 GB/s bandwidth, and was successfuly used for PR applications.
IBM also proposed the Zero Instruction Set Computer (ZISC) [124], their own neural processor. It was composed of a matrix of processing elements that act like a kernel function of an RBF network: as detailed in Section 2.1.1.1, the output value depends on the distance between the input vector and a prelearnt center. Considering the simplicity

[6] Naive Bayes are a class of classification frameworks, the principle of which is to assume each component of the feature vector is indenpendent from the others – hence the word *naive*.

and the parallelism of that architecture, it is easily scalable: for instance the ZISC 36 had a matrix of 36 PEs, while the ZISC 78 contained 78 PEs. Another processor called CM1K, largely inspired by the ZISC, was commercialized by CogniMem Technologies Inc [125]. While the overall principle is similar to the ZISC, it has higher capabilities. Indeed, it embeds 1024 neurons, and is able to take whole images directly as inputs.

Nowadays, several companies are also actively developing embedded solutions based on neural networks. Qualcomm will soon release their next generation of embedded processors, which shall embed a neural coprocessor named Zeroth. That coprocessor is spike-based, and it potentially allows very low power consumption as explained above in Section 2.2.2.1, page 26. It aims to be embedded on mobile platforms, such as mobile phones for instance. Another company, Synopsys, released two processors as Intellectual Property cores (IP cores)[7] optimized for embedded video processing. They are called EV52 and EV54. The IP cores are configurable, and thus the architecture may vary depending on the application. However, its top modules are always the same: a processor with four or two CPUs depending on the model (EV52 or EV54) that performs image processing routines, some memory modules and an object recognition engine, which is basically a matrix of PEs that take care of the computations needed for classification, for instance with a ConvNet. Synopsys claims an energy efficiency 5× better than GPUs, with 1 TOPS/W.

The European Union founded for 10 years a project similar to the Blue Brain project presented earlier, though more oriented towards the simulation of the human brain itself: the Human Brain Project [126]. The project team hopes to gain a better understanding of the human brain, with a twofold benefit: gain the ability to simulate neurodegenerative diseases such as Alzheimer's for medical purpose, and deepen our comprehension of cortical mechanisms to improve neuro-inspired intelligent systems, which encompass the present work. Although the project gave birth to numerous contribution, it is subject to controversies similar to that of the Blue Brain project: scientists found it to cost too much money, and that its goals are unrealistic [127].

2.3 Discussion

In the previous sections of this chapter, the theoretical background of pattern recognition was presented as well as different implementations of pattern recognition framework on different platforms. This Section is dedicated to the comparison of those frameworks. Descriptors and then classifiers shall be discussed in terms of robustness and complexity,

[7] Hardware modules that may be used as black boxes on FPGAs.

with an emphasis on how well they may be embedded. Afterwards the problematics underlying the research work presented here shall be stated.

2.3.1 Descriptors and classifiers comparison

2.3.1.1 Descriptors

Mikolajczyk *et al* proposed in 2005 [128] a thorough and comprehensive evaluation of the main descriptors used at that time. While now somehow outdated, as HMAX and ConvNet were not as developed as they are nowadays, that article is still a useful document as it sets the basis for the evaluation of latter descriptors and their comparison with earlier ones. That article compares notably SIFT, described in Section 2.1.2.1, with methods that are not presented here. They also presented the *Gradient Location and Orientation Histogram* (GLOH) descriptor they created, built upon SIFT. It is shown that GLOH perform best at many image classification task, closely followed by SIFT – as the two descriptors are fundamentally of the same nature, this shows the superiority of SIFT compared to other frameworks of that time. As for the SURF descriptors, its authors claimed in [33] that it was both more accurate and faster than SIFT.

The accuracy brought by HMAX for computer vision was groundbreaking [31]. It showed better performances than SIFT in many object recognition tasks, mainly on the Caltech101 dataset. Those results were corroborated by the work of Moreno *et al*, who compared the performances of HMAX and SIFT on object detection and face localization tasks, and found out that HMAX performed indeed better than SIFT [129]. It is also worth mentionning the very interesting work of Jarett *et al* [130], in which they evaluated the contribution of several properties of different computer vision frameworks applied to object recognition. That paper confirms and generalizes the aforementioned work of Moreno *et al*: it states that multi-stage architectures in general, which includes HMAX and ConvNets, perform better than single-stage ones, such as SIFT.

ConvNet achieves outstandingly good performances on large datasets, such as MNIST [48] or ImageNet [131, 132]. In comparison, HMAX's performances are lower. However, its number of parameters to optimize is very large, therefore a ConvNet needs a huge amount of data to be trained properly – indeed, models with lots of parameters are known to be more subject to overfitting [24]. If the data is sparse, it is worth considering using a framework with less parameters, such as HMAX; as explained in Section 2.1.2.2, its training stage simply consists in cropping images at random locations and scales. Despite the fact that that randomness is clearly suboptimal and has been subject to optimization works in the past [46], it presents the advantage of being very simple.

TABLE 2.2: Comparison of descriptors.

Framework	**Accuracy**	**Training**	**Complexity**
ISCN	High	None	High
HMAX	High	Yes, requires few data points	High
HOG	Reasonnable	None	Low
SIFT	Reasonnable	None	Low
SURF	Reasonnable	None	very low
ConvNet	Very high on large datasets	Yes, requires a large dataset	High

Furthermore, while it has been stated that HMAX's accuracy is related to the amount of features in the S2 dictionnary, the performance do not evolve so much after 1,000 patches. Assuming only 1 patch per image is cropped during training, then one would require 1,000 which is much lower than the tens of thousands usually gathered to train ConvNet [48, 50]. That state of things led to the thought that while working in many situations, ConvNet may not be the most adapted tool for all applications – particularly in the case where the training set is small. Another possibility would be to use an Invariant Scattering Convolution Network as the first layers of a ConvNet, as suggested in [38], instead of optimizing the weights of the convolution kernels during the training stage.

Due to their performances, those three multistage architectures – ConvNet, ISCN and HMAX – seem like the most promising options for most computer vision applications. However, another important aspect that must be taken into account is that of their respective complexities: they have different requirements in terms of computational resources and memory that shall be decisive when choosing one of them, especially in the case of embedded systems. To that respect legacy descriptors such as HOG, SIFT and SURF in particular are interesting alternatives.

In order to set boundaries to the present work, a few descriptors must be chosen so that most of the effort can focus on them. To that end, Table 2.2 sums up the main features of the presented descriptors. As the aim is to achieve state of the art accuracy, the work presented in this thesis shall mostly relate to the three aforementioned multistage architectures: ConvNet, ISCN and HMAX.

2.3.1.2 Classifiers

For a given application, after selecting the (believed) most appropriate descriptor one must choose a classifiers. Like descriptors, they have different features in terms of robustness, complexity and memory print, both for training and prediction. Most of the

time, the classification stage itself is not the most demanding in a processing chain, and thus may not need to be accelerated. In the case where one need such acceleration, the literature on the subject is already substantial – see Section 2.1.1. For those reasons, the present document shall not address hardware acceleration for classification. However, as the choice of the classifier plays a decisive role in the robustness of the system, the useful criteria for classifier selection shall be presented.

Let's first consider the training stage. As it shall be in any case performed on a workstation and not on an embedded system, constraints in terms of complexity and memory print are not so high. However, a clear difference must be made between the *iterative* training algorithms and the others. An iterative algorithm processes the training samples one by one, or by batch – they do not need to load all the data in once, and are therefore well suited for training with lots of samples. On the other hand, non-iterative data such as SVM or RBF need the whole dataset in memory to be trained, which is not a problem for reasonably small datasets but may become one when there are many datapoints – obviously the limit depends on the hardware configuration used to train the machine, though in any case efficient training requires strong hardware.

The classifier must also be efficient during predictions – here, "efficiency" is meant as speed, as the robustness depends largely on training. Feedforward frameworks, as most of those presented here, present the advantage of being fast compared to more complex frameworks. In linear classifiers such as Perceptrons or linear SVMs, the classification often simply consists in a matrix multiplication, which is now well optimized even on non massively parallel architectures like CPUs, thanks to libraries such as LAPACK [133] or BLAS [134].
The speed of kernel machines, e.g RBF or certain types of SVM, is often directly related to the number of used kernel functions. For instance, the more training examples, the more kernels an RBF net may have (see Appendix A). Particular care must therefore be taken during the training stage of such nets, so that the number of kernels stays to a manageable amount. Finally, ensemble learning frameworks such as Boosting algorithms are often used when speed is critical in an applications, and have been demonstrated to be very efficient in the case of face detection for instance [30].

Those considerations put aside, according to the literature HMAX is best used with either AdaBoost or SVM classifiers respectively for one-class and multi-class classification tasks [31]. Concerning ISCN, it is suggested to use a SVM for prediction [38]. Concerning ConvNet, it embeds its own classification stage which typically takes the form of an MLP [48, 49].

Now that the advantages and drawbacks of both the classification and feature extraction frameworks have been stated, the next section proposes a comparison between different implementation techniques.

2.3.2 Implementations comparison

In order to implement those frameworks a naive approach would be to implement them on a CPU, as it is probably the most widespread computing machine. However that would be particularly inefficient, as those frameworks are highly parallel and that such devices are by nature sequential: a program consists in a list of successive instructions that are run one after the other. Their main advantage, however, is that they are fairly easy to program. For that reason, CPU implementations still remain a quasi-mandatory step when testing a framework.

GPUs are also fairly widespread devices, even in mainstream machines. The advent of video games demanding more and more resources dedicated to graphics processing led to a massive production of those devices, which provoked a dramatic drop in costs. For those reasons they are a choice target platforms for many neuromorphic applications. While somewhat more complicated to program that CPUs, the coming of higher level languages such as CUDA made the configuration of GPU reasonably easy to reach. The amount of frameworks using that kind of platforms, and moreover their success show that it is a very popular piece of hardware for that purpose [53, 56–58, 135]. However, their main disadvantage is their volume and power consumption, the latter being in the order of magnitude of 10 W. For embeddable systems the power consumption should not go beyond 1 W, which is where reconfigurable hardware devices are worth considering.

FPGAs present two major drawbacks: they are not as massively produced as GPUs and CPUs, which raises their cost. Their other downside actually goes alongside with their highest quality: they are entirely reconfigurable, from the way the computations are organized to the data coding scheme and such flexibility comes to the price of a higher development time (and thus cost) than CPUs and GPUs. However their power consumption is most of the time below 1 W, and can be optimized with appropriate coding principles. They are also much smaller than GPUs, and the low power consumption leads to colder circuits, which allows to save the energy and space that would normally be required to keep the device at a reasonnable temperature. Furthermore, they are reconfigurable to a much finer grain than GPU, and thus provide even more parallelization as the latters. All these criteria make FPGAs good candidates for embedded implementations of computer vision algorithms.

2.3.3 Problem statement

The NeuroDSP project presented in Section 1.4 aims to propose an integrated circuit for embedded neuromorphic computations, with high constraints in terms of power consumption, volume and cost. The ideal solution would be to produce the device as an Application Specific Integrated Circuit (ASIC) – however its high cost makes it a realistic choice only in the case where the chip is guaranteed to be sold in high quantities, which may be a bit optimistic for a first model. For that reason, we chose to implement that integrated circuit on an FPGA. As one of the aim of NeuroDSP is to be cost-efficient, we aim to propose those neuromorphic algorithms on mid-range hardware. Towards that end, one must optimize them w.r.t two aspects: complexity and hardware resource consumption. The first aspect may be optimized by identifying what part of the algorithm is the most important, and what part can be discarded. A way to address the second aspect of the problem is to optimize data encoding, so that computations on them requires less logic. Those considerations lead to to the following problematics, which shall form the matter of the present document:

- **How may neuromorphic descriptors be chosen appropriately and how may their complexity be reduced?**
- **How the data handled by those algorithms may be efficiently coded so as to reduce hardware resources?**

2.4 Conclusion

In this chapter we presented the works related to the present document. The problematics that we aimed to address were also stated. The aim of the contributions presented here is to implement efficient computer vision algorithms on embedded devices, with high constraints in terms of power consumption, volume, cost and robustness. The primary use case scenario concerns image classification tasks.

There exist many theoretical frameworks allowing to classify data, be it images, one dimensional signals or other. Naive algorithms such as Nearest Neighbor have the advantage of being really simple to implement; however they may achieve poor classification performances, and cost too much memory and computational power when used on large datasets. More sophisticated frameworks, such as neural networks, SVMs or ensemble learning algorithms can achieve better results.
In order to help the classifier, it is also advisable to use a descriptor, the aim of which is to extract data from the sample to be processed. Among such descriptors figures HMAX,

which is inspired by neurophysiological data acquired on mamals. Such frameworks are said to be neuro-inspired, or bio-inspired. Another popular framework is ISCN, which decomposes the input image with particular filters called wavelets.
One of the most popular frameworks nowadays is ConvNet, which is a basically a classifier with several preprocessing layers that act as a descriptor. While impressively efficient, it needs to be trained with a huge amount of training data, which is a problem for applications where data is sparse. In such case it may seem more reasonable to use other descriptors, such as HMAX or ISCN, in combination with a classifier.

The algorithms mentioned above are most of the time particularly well suited for parallel processing. While it is easier to implement them on CPU using languages such as C, the efficiency gained when running them on massively parallel architecture makes it worth the effort. There exist several frameworks using GPU acceleration, however GPUs are ill-suited for most embedded applications where power consumption is critical. FPGAs are better candidates in those cases, and contributions about implementations on such devices have been proposed.

The aim of the work presented in this document is to implement those demanding algorithms on mid-range reconfigurable hardware platforms. To achieve that, it is necessary to adapt them to the architecture. Such study is called “Algorithm-Architecture Matching” (AAM). That need raises two issues: how those frameworks may be reduced, and how the data handled for computation may be efficiently optimised, so as to use as few hardware resources as possible? The present document proposes solutions addressing those two questions.

Chapter 3

Feature selection

This chapter addresses the first question stated in Chapter 2, concerning the optimizations of a descriptor for specific applications. The first contribution presented here is related to a face detection task, while the second one proposes optimizations adapted to a pedestrian detection task. In both cases, the optimization scheme and rational are presented, along with a study of the complexity of major frameworks addressing the considered task. Accuracies obtained with the proposed descriptors are compared to those obtained with the original framework and the described systems of the literature. Those changes in accuracies are then put in perspectives with the computational gain. General conclusions are presented at the end of this Chapter.

3.1 Feature selection for face detection

This Section focuses on a handcrafted feature extractor for a face detection application. We start from a descriptor derived from HMAX, and we propose a detailed complexity analysis; we also determine where lies the most crucial information for that specific application, and we propose optimizations allowing to reduce the algorithm complexity. After reminding the reader of the major techniques used in face detection, we present our contribution, which consists in finding and keeping the most important information extracted by a framework derived from HMAX. Performance comparison with state of the art frameworks are also presented.

3.1.1 Detecting faces

For many applications, being either mainstream or professional, face detection is a crucial issue. Its more obvious use case is to address security problems, e.g identifying a person

may help in deciding whether access should be granted or denied. It may also be useful in human-machine interactions, for instance if a device should answer in some way in case a human user shows particular states, such as distress, pain or unconsciousness – and to do that, the first step is to detect and locate the person's face. In that second scenario we fall into embedded systems, which explains our interest in optimizing face detection frameworks. Among the most used face detection techniques lie Haar-like feature extraction, and as usual ConvNet. We shall now describe the use of those two paradigms in those particular problems, as well as a framework called HMIN which is the basis of our work.

3.1.1.1 Cascade of Haar-like features

Before the spreading of ConvNets, one of the most popular framework for face detection was the Viola-Jones algorithm [30, 136] – it is still very popular, as it is readily implemented in numerous widely used image processing tools, such as OpenCV [137]. As we shall see, the main advantage of this framework is its speed, and its decent performances.

Framework description Viola's and Jones' framework is built along two main ideas [136]: using easy and fast to compute low-level features – the so-called Haar-like features – in combination with a Boosting classifier that selects and classifies the most relevant features. Classifiers are cascaded so that the most obviously not-face regions of the image are discarded first, allowing to spend more computational time on most promising regions. A naive implementation of the Haar-like features may use convolution kernels, consisting of 1 and -1 coefficients, as illustrated on Figure 3.1. Such features may be computed efficiently using an image representation proposed in [30, 136] called *Integral Image*. In such representation, the pixel located at (x, y) takes as value the sum of the original image's pixels located in the rectangle defined by the $(0, 0)$ and the (x, y) point, as shown in Figure 3.2. To compute such an image F one may use the following recurrent equation [30]:

$$F(x, y) = F(x - 1, y) + s(x, y), \tag{3.1}$$

with

$$s(x, y) = s(x, y - 1) + f(x, y) \tag{3.2}$$

where $f(x, y)$ is the original image's pixel located at (x, y). Using this representation, the computation of a Haar-like feature may be performed with few addition and subtraction operations. Moreover the number of operations does not depend on the scale of the considered feature. Let's consider first the feature on the left of Figure 3.1, and let's

FIGURE 3.1: Example of Haar-like features used in Viola-Jones for face detection. They can be seen as convolution kernels where the grey parts correspond to $+1$ coefficients, and the white ones -1. Such features can be computed efficiently using integral images [30, 136]. Point coordinates are presented here for latter use in the equations characterizing feature computations.

assume its top-left corner location is (x_1, y_1) and that of its bottom-right corner's is (x_2, y_2). Given the integral image II, its response $r_l(x_1, y_1, x_2, y_2)$ is given by

$$r_l(x_1, y_1, x_2, y_2) = F(x_1, y_g, x_2, y_2) - F(x_1, y_1, x_2, y_g) \tag{3.3}$$

with $F(x_1, y_1, x_2, y_2)$ the integral of the values in the rectangle delimited by (x_1, y_1) and (x_2, y_2), expressed as

$$F(x_1, y_1, x_2, y_2) = II(x_2, y_2) + II(x_1, y_1) - II(x_1, y_2) - II(x_2, y_1) \tag{3.4}$$

where $II(x, y)$ is the value of the integral images at location (x, y). As for the response $r_r(x_1, y_1, x_2, y_2)$ of the feature on the right, we have:

$$r_r(x_1, y_1, x_2, y_2) = F(x_w, y_2, x_g, y_1) - F(x_1, y_1, x_g, y_2) - F(x_w, y_1, x_2, y_2) \tag{3.5}$$

The locations of the points are shown in Figure 3.1. Once features are computed, they are classified using a standard classifier such as a perceptron for instance. If the classifier does not reject the features as "not-face", complementary features are computed and classified, and so on until either all features are computed and classified as "face", or the image is rejected. This cascade of classifiers allows to reject most non-faces images early in the process, which is one of the main reasons for its low complexity.

Now that we described the so-called Viola-Jones framework, we shall study its computational complexity.

Complexity analysis Let's now evaluate the complexity involved by that algorithm when classifying images. The first step of the computation of those Haar-like features on an image is then to compute its integral image. According to Equation 3.1 and Equation 3.2, it takes only 2 additions per pixels. Then, the complexity $C_{\mathrm{VJ_{II}}}$ of this

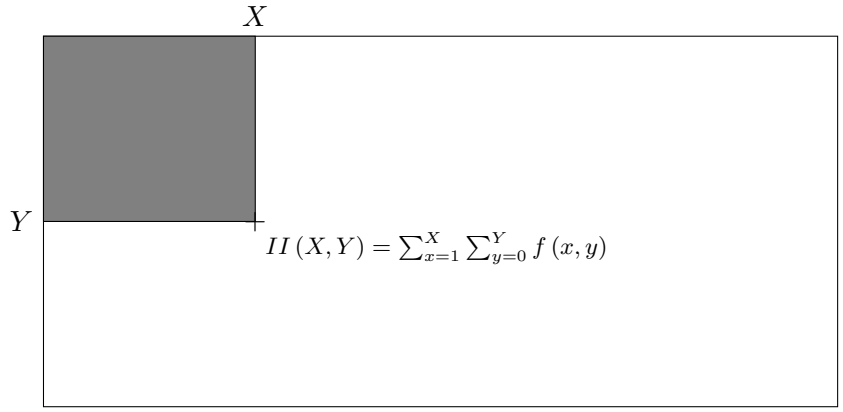


FIGURE 3.2: Integral image representation. $II(X,Y)$ is its value of the point coordinated (X,Y), and $f(x,y)$ the value of the original image at location (x,y) [30].

process for a $w \times h$ image is given by

$$C_{\mathrm{VJ_{II}}} = 2wh. \tag{3.6}$$

That serves as the basis of the computation of the Haar-like features, as we saw earlier. The complexity highly depends on the number of computed features, and for this study we shall stick to the implementation proposed in the original paper [136]. In that work, the authors have a total a 6060 features to compute – however, they also claimed that, given the cascade of classifiers they used, only $\overline{N_f} = 8$ features are computed in average. From [136], we now that each feature needs from 6 to 9 operations to compute – we shall consider here that, on average, they need $N_{\mathrm{op}} = 7.5$ operations. We note that, thanks to the computation based on the integral image, the number of operations does not depend on the size of the computed feature. After that, the features are classified – however we focus our analysis on the feature extraction only, so we do not take that aspect into account here. Thus, denoting C_{VJ_F} the complexity involved a this stage, we have

$$C_{\mathrm{VJ}_F} = \overline{N_{\mathrm{op}}} N_f. \tag{3.7}$$

In additions, images must be normalized before being processed. Viola *et al.* proposed in [136] to normalize the contrast of the image by using its standard deviation σ given by

$$\sigma = m^2 - \frac{1}{N} \sum_{i=0}^{N} {x_i}^2, \tag{3.8}$$

where m is the mean of the pixels of the image, $N = wh$ is the number of pixels and x_i is the value of the i-th pixel. Those values may be computed simply as

$$m = \frac{II\,(W,H)}{wh} \tag{3.9}$$

$$\frac{1}{N}\sum_{i=0}^{N} {x_i}^2 = \frac{II^2\,(W,H)}{wh} \tag{3.10}$$

where II^2 denotes is the integral image representation of the original image with all its pixels squared. The computation of that integral image needs thus one power operations per pixel, to which we must add the computations required by the integral images, which leads to a total of $3WH$ operations. Computing m requires a single operation, as computing $\frac{1}{N}\sum_{i=0}^{N} {x_i}^2$. As the feature computation is entirely linear and since the normalization simply consists in multiplying the feature by the standard deviation, that normalization may simply be applied after the feature computation, involving a single operation per feature. Thus, the complexity C_{VJ_N} involved by image normalization is given by

$$C_{\mathrm{VJ}_N} = 3wh + N_f \tag{3.11}$$

From Equations 3.6, 3.7 and 3.11, the framework's global complexity is given by

$$\begin{aligned} C_{\mathrm{VJ}} &= C_{\mathrm{VJ}_{\mathrm{II}}} + C_{\mathrm{VJ}_F} + C_{\mathrm{VJ}_N} \\ &= 5wh + \left(\overline{N_{\mathrm{op}}} + 1\right) N_f, \end{aligned} \tag{3.12}$$

which considering the implementation proposed in [136], i.e with $w = h = 24$ and $N_f = 7.5$, leads to a total of 2948 operations. Although strikingly low, it must be emphasized here that that value is an *average*; when a face is actually detected, all 6060 features must be computed and classified, which then leads to 54,390 operations. However, for fair comparison we shall stick to the average value latter in the document.

Now that we evaluated the complexity of the processing of a single $w \times h$ image, let's evaluate it in the case where we scan a scene in order to find and locate faces. Normally, one would typically use several sizes of descriptors in order to find faces of different sizes [50, 136] – however, in order to simplify the study we shall stick here to a single scale. Let W and H respectively be the width and height of the frame to process, and let N_w be the number of windows processed in the image. If we classify subwindows at each location of the image, we have

$$N_w = (W - w + 1)\,(H - h + 1) \tag{3.13}$$

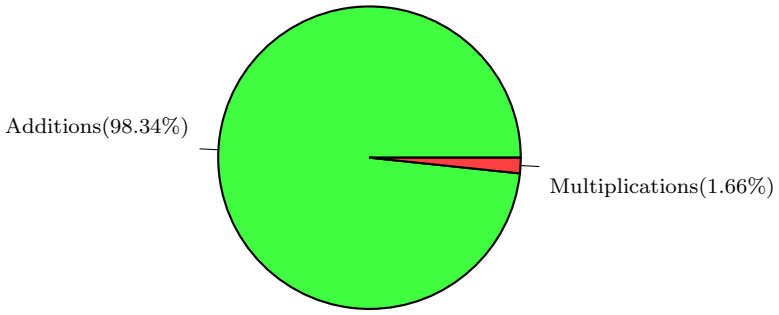


FIGURE 3.3: Complexity repartition of Viola and Jones' algorithm when processing a 640×480 with a 24×24 sliding window. From Equations 3.7 to 3.13, we see that the integral image computation requires $2WH$ additions, the feature extraction needs $\overline{N_{\text{op}}} N_f N_w$ additions, and C_{VJ_N} needs WH multiplications and $2WH$. Thus, we need a total of $4WH + \overline{N_{\text{op}}} N_f N_w$.

The integral images are first computed on the whole 640×480 image; after that, features must be computed, normalized and classified for each window. From Equations 3.6, 3.7, 3.11 and 3.13 we know that we need

$$C_{\text{VJ}} = 2WH + \overline{N_{\text{op}}} N_f N_w + N_f N_w \tag{3.14}$$

$$= 2WH + N_f N_w \left(\overline{N_{\text{op}}} + 1\right) \tag{3.15}$$

$$= 5WH + N_f \left(W - w + 1\right)\left(H - h + 1\right)\left(\overline{N_{\text{op}}} + 1\right). \tag{3.16}$$

In the case of a 640×480 image, with $w = 24$, $h = 24$, $N_f = 8$ and $\overline{N_{\text{op}}} = 7.5$ as before, we get $C_{\text{VJ}} = \mathbf{20.7}$ MOP. Figure 3.3 shows the repartition of the complexity into several types of computations, considering that we derive from the above analysis that we need $4WH + \overline{N_o p} N_f$ additions and WH multiplications.

Memory print Let's now evaluat the memory required by that framework when processing a 640×480 image. Assuming the pixels of the integral image are coded on 32 bits integers, the integral image would require 1.2 MB to be stored entirely. Assuming ROIs are evaluated sequentially on the input image, 6060 features are computed at most and each feature is coded as 32-bits integers, we would require 24.24 ko to stores the features. Thus, the total memory print required by that framework would be, in that case, 1.48 MB. That framework also has the great advantage that a single integral image may be used to compute features of various scales, without the need of computing, storing and managing an image pyramid, as required by other frameworks – more information about image pyramids are available in Section 3.1.3.2.

We presented the use of Haar-like features in combination with the AdaBoost classifier for face detection task, proposed by Viola and Jones [30, 136]. We shall now present and analyse an other major tool for this task, which is called CFF.

3.1.1.2 Convolutional Face Finder

Framework description Since they show impressive performance in many domains, it is only natural that implementations of ConvNet were proposed for face detection applications [50]. One of the main implementations is the Convolutional Face Finder (CFF). It is composed of 6 layers: C1, S1, C2, S2, N1 and N2. CX denotes a convolution layer, SX a subsampling layer and N a partially connected layer. It takes 32×36 images as inputs. The C1 layer consists in 4 convolutions with 5×5 kernels, thus producing 28×32 patches which are downsampled to 14×16 patches in S1. At that stage, the downsampling strategy for each unit in S1 consists in computing the average of 2×2 features from a single C1 feature map, to multiply it by a coefficient, then to add a bias to the result and to apply a non-linearity. Two neighboring units in S1 have contiguous, non-overlapping receptive fields. The resulting S1 feature map feed then C2, where they are filtered again using convolutions: each of the four feature maps produced in S1 feeds two convolution kernels, and 6 pairs of those same S1 feature maps feed another convolution kernel each, which leads to a total of 14 feature maps of size 12×14. All those convolution kernel are 3×3. The produced 14 feature maps are then downsampled in S2 to produce 6×7 feature maps, using the same strategy as in S1 but this times with 3×3 receptive fields.
N2 has 14 units, one per S2 feature maps, where each unit is fully connected to the units of a single S2 feature map. Thus, each N1 unit is connected to 168 S2 units. Finally, in N2 all N1 units are connected to a single output unit, that gives the classification result. The framework is shown in Figure 3.4.

During the prediction stage, it should be noted that the network can in fact process the whole image at once, instead of running the full computation windows by windows [48, 138]. This technique allows to save lots of computations, and is readily implemented if one considers the N1 layer as a convolution filter bank with kernel of size 6×7, and the N2 layer like another filter bank with 1×1 convolution kernels [139].

Complexity analysis Let's now evaluate the complexity involved by the CFF algorithm. Denoting $C_{\text{CFF}_{XX}}$ the complexity brought by the layer XX, and neglecting the classification as done in Section 3.1.1.1, we have

$$C_{\text{CFF}} = C_{\text{CFF}_{C1}} + C_{\text{CFF}_{S1}} + C_{\text{CFF}_{T1}} + C_{\text{CFF}_{C2}} + C_{\text{CFF}_{S2}} + C_{\text{CFF}_{T2}}, \tag{3.17}$$

where TX represents a non-linearity layer, where an hyperbolic tangeant is applied to each feature of the input feature map. Let's first evaluate $C_{\text{CFF}_{C1}}$. It consists in 4 convolutions, which consists mainly in Multiplication-Accumulation (MAC), which we

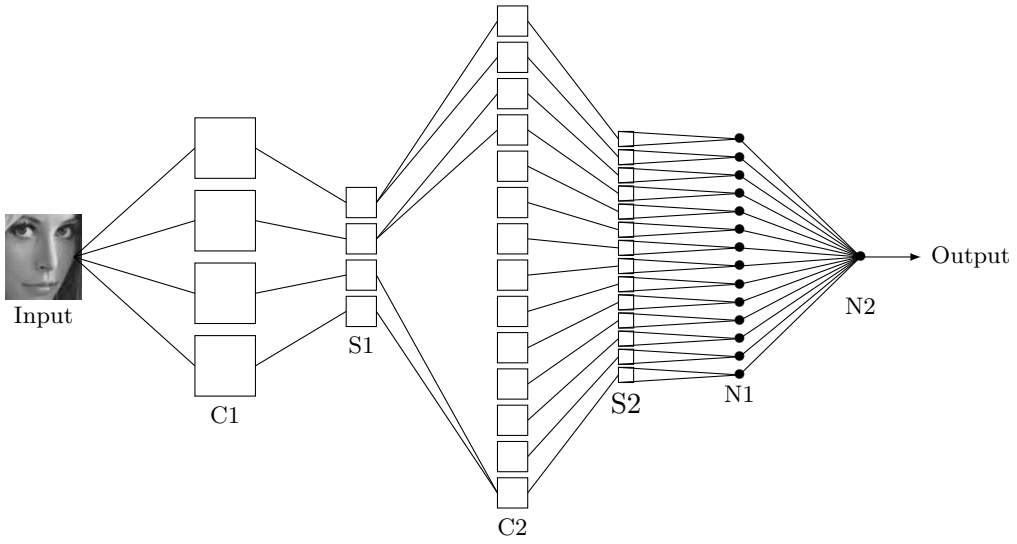


FIGURE 3.4: Convolutional Face Finder [50]. This classifier is a particular topology of a ConNet, consisting in a first convolution layer C1 having four trained convolution kernels, a first sub-sampling layer S1, a second convolution layer C2 partially connected to the previous layer's units, a second sub-sampling layer S2, a partially-connected layer N1 and a fully-connected layer N2 with one output unit.

assume corresponds to a single operation as it may be done on dedicated hardware. Thus we have

$$C_{\mathrm{CFF}_{\mathrm{C1}}} = 4 \times 5 \times 5 \,(W-4)\,(H-4) \tag{3.18}$$

$$= 100WH - 400\,(W+H) + 1600. \tag{3.19}$$

Since the S2 layer consists in the computation of means of features in contiguous non-overlapping receptive fields, this means that each feature is involved once an only once in the computation of a mean, which also requires a MAC operation per pixel. Considering that at this point, we have 4 $(W-4) \times (H-4)$ feature maps, and so

$$C_{\mathrm{CFF}_{\mathrm{S1}}} = 4\,(W-4)\,(H-4) \tag{3.20}$$

$$= 4WH - 16\,(W+H) + 64. \tag{3.21}$$

Now, the non-linearity layer must be applied: an hyperbolic tangeant function is used to each feature of the 4 $W_{\mathrm{S2}} \times H_{\mathrm{S2}}$ feature maps, with

$$W_{\mathrm{S2}} = \frac{W-4}{2} \tag{3.22}$$

$$H_{\mathrm{S2}} = \frac{H-4}{2}, \tag{3.23}$$

and thus, considering the best case where an hyperbolic tangent may be computed in a single operation,

$$C_{\mathrm{CFF}_{T1}} = 4\left(\frac{W-4}{2}\right)\left(\frac{H-4}{2}\right) \tag{3.24}$$

$$= WH - 2(W+H) + 16 \tag{3.25}$$

The C2 layers consists in 20 convolution, the complexity of which may be derived from 3.18. Then, there are 6 element-wise sums of feature maps, which after the convolutions are of dimensions

$$\left(\frac{W-4}{2} - 2\right) \times \left(\frac{H-4}{2} - 2\right), \tag{3.26}$$

and thus we have

$$C_{\mathrm{CFF}_{\mathrm{C2}}} = (20 \times 3 \times 3 + 6 \times 3 \times 3)\left(\frac{W-4}{2} - 2\right)\left(\frac{H-4}{2} - 2\right) \tag{3.27}$$

$$= 9 \times 26\left(\frac{W}{2} - 4\right)\left(\frac{H}{2} - 4\right) \tag{3.28}$$

$$= 234\left(\frac{WH}{4} - \frac{5}{2}(W+H) + 16\right) \tag{3.29}$$

$$= 58.5WH - 585(W+H) + 3744. \tag{3.30}$$

The complexity in S2 layer may be derived from Equations 3.20 and 3.26, giving

$$C_{\mathrm{CFF}_{\mathrm{S2}}} = 3.5WH - 28(W+H) + 224. \tag{3.31}$$

And finally, the complexity of the last non-linearity may be expressed as

$$C_{\mathrm{CFF}_{\mathrm{T2}}} = 14W_{\mathrm{S2}}H_{\mathrm{S2}} \tag{3.32}$$

with

$$W_{\mathrm{S2}} = \frac{1}{2}\left(\frac{W-4}{2} - 2\right) \tag{3.33}$$

$$H_{\mathrm{S2}} = \frac{1}{2}\left(\frac{H-4}{2} - 2\right), \tag{3.34}$$

which gives

$$C_{\mathrm{CFF}_{\mathrm{T2}}} = 1.75WH + 7(W+H) + 16. \tag{3.35}$$

Using those results in Equation 3.17, we finally get

$$C_{\mathrm{CFF}} = 168.75WH - 1038(W+H) + 5664. \tag{3.36}$$

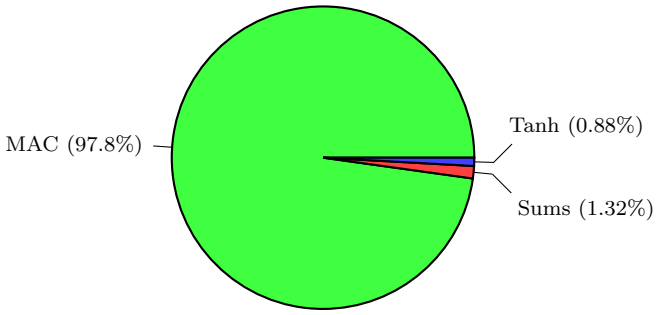


FIGURE 3.5: Complexity repartition of the CFF algorithms, separated in three types of computations: MAC, hyperbolic tangents ("Tanh") and sums. We see here that the large majority of operations are MAC, toward which most effort should then be put for fine optimizations or hardware implementation.

Now that we have this general formula, let's compute the complexity involved in the classification of a typical 36×32 patch. We get 129.5 kOP. Let's now assume that we must find and locate faces in a VGA 640×480 image. From [48] and [50] and as recalled earlier, we know that the features may be efficiently extracted at once in the whole image, by applying all the convolutions and subsampling directly to it. Thus, we may compute that complexity directly by reusing Equation 3.36, and we get **50.7 MOP**. Figure 3.5 shows the repartition of the complexity of that frameworks.

Memory print Let's now evaluate the memory required by the CFF framework. As in Section 3.1.1.1, we shall consider here the case where we process a 640×480 image, without image pyramid. The first stage produces 4 636×476 feature maps – assuming the values are coded using single precision floating point scheme, hence using 32 bits, that stage requires a total of 4.84 MB. As the non-linearity and subsampling stages may be performed in-place, they do not bring any further need in memory. The second convolution stage, however, produces 20 316×236 feature maps. Using the same encoding scheme as before, we need 59.7 MB. We should also take into account the memory needed by the weights of the convolution and subsampling layers, but it is negligible compared to the values obtained previously. Hence, the total memory print is 64.54 MB. It should by noted that that amount would be much higher in the case where we process an image pyramid, as usually done. However, we stick to an evaluation on a single scale here for consistency with the complexity study.

This Section was dedicated to the description and study of the CFF framework. Let's now do the same study on another framework to which we refer as HMIN.

3.1.1.3 HMIN

Framework description In order to detect and locate faces on images, one may use HMAX, which was described in Section 2.1.2.2. However using that framework to locate an object requires to process separately different ROI of the image. In such case, the S2 and C2 layers of HMAX provide little gain in performance, as it is mostly useful for object detection in clutter [31]. Considering the huge gain in computation complexity when not using the two last layers, we propose here to use only the first two layers for our application. In the rest of the document, the framework constituted by the S1 and C1 layers of HMAX shall be referred to as *HMIN*.

We presented the so-called framework HMIN, on which we base our further investigations. We shall now study its complexity, along the lines of what we have proposed earlier for Viola-Jones and the CFF.

Complexity analysis The overall complexity C_{HMIN} involved by the two stages S1 and C1 of HMIN is simply

$$C_{\text{HMIN}} = C_{\text{HMIN}_{\text{S1}}} + C_{\text{HMIN}_{\text{C1}}} \tag{3.37}$$

Where $C_{\text{HMIN}_{\text{S1}}}$ and $C_{\text{HMIN}_{\text{C1}}}$ are respectively the complexity of the S1 and C1 layers. The S1 layer consists in a total of 64 convolutions on the $W \times H$ input image. Different kernel sizes are involved, but it is important that all feature maps fed in the C1 layer are of the same size. Thus, the convolution must be computed at all positions, even those where the center of the convolution kernel is on the edge of the image. Missing pixel may take any value: either simply 0 or the value of the nearest pixel for instance. Denoting k_i the size of the convolution kernel at scale i presented in the filter column of Table 2.1, we may write

$$C_{\text{HMIN}_{\text{S1}}} = 4\sum_{i=1}^{16} WHk_i{}^2 = 36146WH. \tag{3.38}$$

As for the C1 layer, it may be applied as follows: first, the element-wise maximum operations accross pairs of feature maps are computed, which take 8WH operations; then we apply the windowed max pooling. Since there is a 50% overlap between the receptive fields of two contiguous C1 units, neglecting the border effects each feature of each S1 feature map is involved in 4 computations of maximums. Since those operations are computed on 32 feature maps, and adding the complexity of the first computation, we get

$$C_{\text{HMIN}_{\text{C1}}} = 8WH + 8 \times 4WH = 40WH. \tag{3.39}$$

From Equations 3.37 to 3.39, we get

$$C_{\text{HMIN}} = 36456WH. \tag{3.40}$$

If we aim to extract feature from a typical 128×128 image for classification as suggested in [31], it needs 597 MOP operation. When scanning a 640×480 image as done with the CFF in Section 3.1.1.2, we get a total of **11.2 GOP**. From Equations 3.38 and 3.39, we also see that the convolutions operations take 99.89 % of the computation – thus, they represent clearly the basis of our optimizations.

Memory print Using the same method as for Viola-Jones' in Section 3.1.1.1 and the CFF in Section 3.1.1.2, let's evaluate the memory print of HMIN. Since the C1 layer may be processed in-place, the memory print of HMIN is the same as its S1 layer, which produces 16 640×480 feature maps, coded on 32-bit single precision floating point numbers. Hence, its memory print is 19.66 MB.

This Section was dedicated to the presentation of several algorithms suited for face detection, include HMIN which shall serve as the basis of our work. Next Section is dedicated to our contributions in the effort of optimizing out HMIN.

3.1.2 HMIN optimizations for face detection

In this Section we propose optimizations for HMIN, specific to face detection applications. We begin by analysing the output of the C1 layer, and we then propose our simplifications accordingly. Experimental results are then shown. This work is based on the one presented in [140], which we pushed further as described below.

3.1.2.1 C1 output

As HMIN intends to be a general purpose descriptor, it aims to grasp features of as various types. Figure 3.6 shows an examples of the C1 feature maps for a face. The eyes, nose and mouth are the most prominent object of the face, and as such one can expect HMIN to be particularly sensible to them as it is based on the mammal's vision system, which can indeed easily be seen in Figure 3.6. One can also see that the eyes and mouths are more salient when $\theta = \pi/2$, and that the nose is more salient when $\theta = 0$. Furthermore, one can also see that the extracted features are redundant from C1 maps of neighboring scales and same orientations.

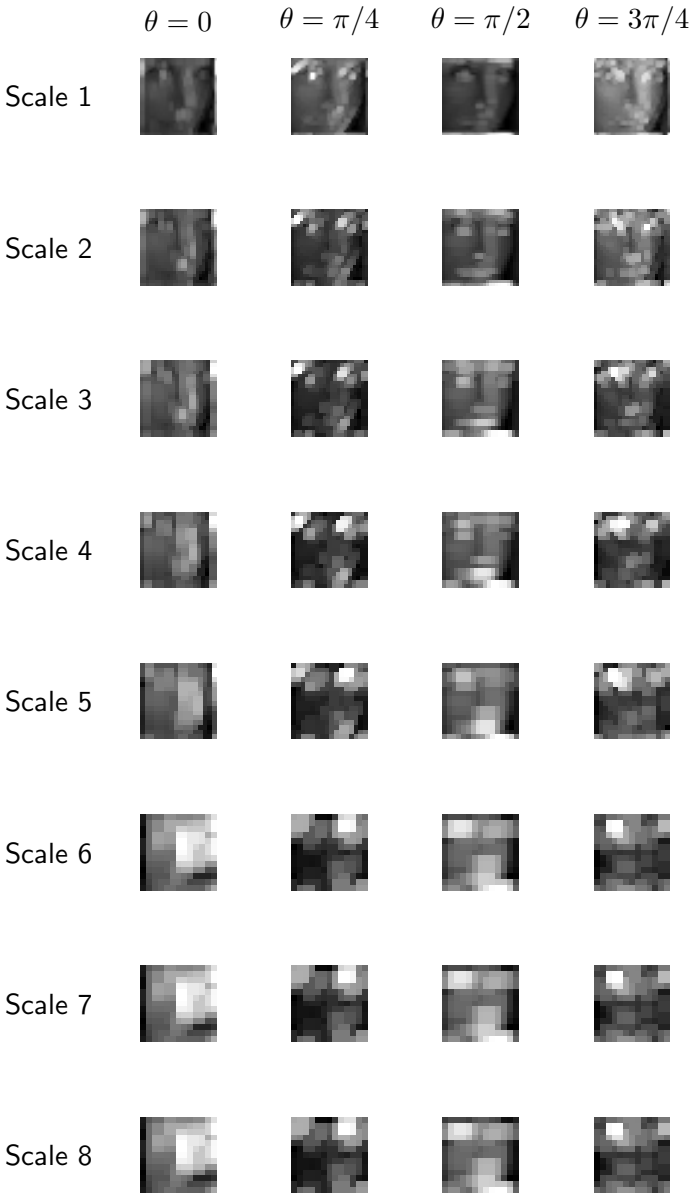


FIGURE 3.6: C1 feature maps for a face. One can see here that most of the features corresponding to an actual feature of a face, e.g the eyes or the mouth, is given by the filters with orientation $\theta = \pi/2$.

3.1.2.2 Proposed optimizations

HMIN$_{\theta=\pi/2}$ From the shown results, we propose to keep only the filters with $\theta = \pi/2$. Due to the redundancy, we also propose to sum the output of the S1 layer – which is equivalent to sum the remaining kernels of the filter bank to produce one, unique 37×37 convolution kernel. The smaller kernels are padded with zeros so that they are all 37×37 and may be sum across coefficients. This operation is sum-up in Figure 3.7. Figure 3.8 also show the output of that unique kernel applied to the image of a face.

Since we now only have one feature map, we must adapt the C1 layer. As all C1 units now

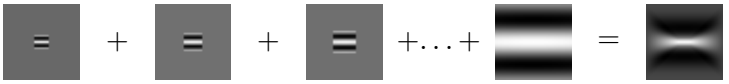


FIGURE 3.7: S1 convolution kernel sum. Kernels smaller that 37×37 are padded with 0's so that they all are 37×37. Kernels are then sum element-wise so as to produce the kernel on the right. It is worth mentioning the proximity of that kernel with one of the feature selected by the Adaboost algorithm in the Viola-Jones framework [30], shown in Figure 3.1.

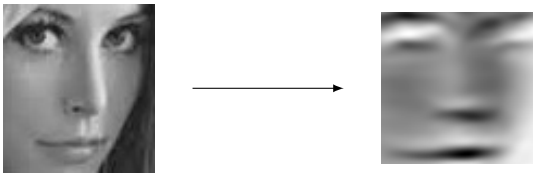


FIGURE 3.8: Feature map obtained with the unique kernel in S1 presented in Figure 3.7. One can see that the eyes mouth and even nostrils are particularly salient.

pool over the only remaining scale, we propose to take the median value N_m among the N_s showed in Table 2.1, namely 16, as the width of the pooling window. Following the lines of the original model, the overlap between the receptive fields of two neighbouring C1 unit shall be $\Delta_m = 8$. We shall refer to this descriptor as $\text{HMIN}_{\theta=\pi/2}$ later on.

Let's now evaluate the complexity involved in this model. We have a single $K \times K$ convolution kernel, with $K = 37$. Applying it to a $W \times H$ image thus requires an amount of MAC operations given by

$$C_{S1} = (W - K - 1)(H - K - 1). \tag{3.41}$$

As for the C1 layer, it needs

$$C_{C1} = (W - K - 1)(H - K - 1) \tag{3.42}$$

maximum operations.

As for the memory print, since we produce a single $(W - K - 1) \times (H - K - 1)$ feature map of single precision floating point numbers, that optimized version of HMIN needs $4(W - K - 1) \times (H - K - 1)$ bytes.

$\text{HMIN}^R_{\theta=\pi/2}$ Following what has been done in earlier, we propose to reduce even further the algorithmic complexity. Indeed, we process somewhat "large" 128×128 face images with a large 37×37 convolution kernel. Perhaps we do not need such a fine resolution – in fact, the CFF takes very small 32×36 images as inputs. Thus, we propose to divide the complexity of the convolution layer by 16 by simply resizing the

convolution kernel to 9×9 using a bicubic interpolation, thanks to Matlab's `imresize` function, with the default parameters. Finally, the maximum pooling layer is adapted by divided its parameters also by 4: the receptive fields are 4×4, with 2×2 overlaps between two receptive fields. Hence, our new descriptor, which we shall refer to as $\text{HMIN}^{R}_{\theta=\pi/2}$ later on, expects 32×32 images as inputs, thus providing vector of the exact same dimensionality than $\text{HMIN}_{\theta=\pi/2}$. The complexity involved by that framework is expressed as

$$C_{\text{HMIN}} = C_{\text{HMIN}_{\text{S1}}} + C_{\text{HMIN}_{\text{C1}}}, \tag{3.43}$$

with

$$C_{\text{HMIN}_{\text{S1}}} = 9 \times 9 \times W \times H = 81WH \tag{3.44}$$

$$C_{\text{HMIN}_{\text{C1}}} = 4WH, \tag{3.45}$$

which leads to

$$C_{\text{HMIN}} = 85WH. \tag{3.46}$$

As we typically expect 32×32 images as inputs, the classification of a single image would take 82.9 kOP. For extracting features of a 640×480 as done previously, that would require 26.1 MOP, and the memory print would be the same as for $HMIN_{\theta=\pi/2}$ assuming we can neglected the memory needed to store the coefficients of the 9×9 kernel, hence we need here 1.22 MB.

3.1.3 Experiments

3.1.3.1 Test on LFWCrop_grey

In this Section, we evaluate the different versions of HMIN presented in the previous Section. To perform the required tests, face images were provided by the Cropped Labelled Face in the Wild (LFW_crop) dataset [141], which shall be used as positive examples. Negative examples were obtained by cropping patches from the "background" class – which shall be refered to as "Caltech101-background" – of the Caltech101 dataset [142] at random positions. All feature vectors $v = (v_1, v_2, \ldots, v_N)$ are normalized so that the lower value is set to 0, and the maximum value is set to 1 to produce a vector $\overline{v} = (\overline{v_1}, \overline{v_2}, \overline{\ldots}, \overline{v_n})$

$$\forall i \in \{1, \ldots, N\} \quad \overline{v_i} = \frac{\dot{v_i}}{\max_{k \in \{1,\ldots,N\}} \dot{v_k}} \tag{3.47}$$

$$\forall i \in \{1, \ldots, N\} \quad \dot{v_i} = v_i - \min_{k \in \{1,\ldots,N\}} v_k \tag{3.48}$$

Descriptor	HMIN	$\text{HMIN}_{\theta=\pi/2}$	$\text{HMIN}^{R}_{\theta=\pi/2}$
Accuracy (%)	95.78 ± 0.97	90.81 ± 1.10	90.05 ± 0.98

TABLE 3.1: Accuracies of the different version of HMIN on the LFW_crop dataset.

For each version of HMIN, we needed to train a classifier. We selected 500 images at random from LFW_crop and another 500 from Caltech101-background. We chose to use an RBF classifier. The images were also transformed accordingly to the descriptor, i.e resized to 128×128 for both HMIN and $\text{HMIN}_{\theta=\pi/2}$ and resized to 32×32 images for $\text{HMIN}^{R}_{\theta=\pi/2}$. The kerneling parameter of the RBF network was set to $\mu = 2$ – see Appendix A for more information about the RBF learning procedure that we used.

After training, 500 positive and 500 negative images were selected at random among the images that were not used for training to build the testing set. All images were, again, transformed w.r.t the tested descriptor, the feature vectors were normalized and classification was performed. Table 3.1 shows the global accuracies for each descriptor, using a naive classification scheme with no threshold in the classification function. Figure 3.9 shows the Receiver Operating Characteristic curves obtained for all those classifiers on that dataset. In order to build those curves, we apply the classification process to all testing images, and for each classification we compare its *confidence* to a threshold. That *confidence* is the actual output of the RBF classifier, and indicates how certain the classifier is that its prediction is correct. If the confidence is higher than the threshold, then the classification is kept; otherwise it is rejected. By modifying that threshold, we make the process more or less tolerant. If the network is highly tolerant, then it shall tend to produce higher false and true positive rates; if it is not tolerant, then on the contrary it shall tend to produce lower true and false positive rates. The ROC curves show how the true positive rate evolve w.r.t the false positive rate.

3.1.3.2 Test on CMU

The CMU Frontal Face Images [143] dataset consists in grayscale images showing scenes with one or several persons (or characters) facing the camera or sometimes looking slightly away. Sample images are presented in Figure 3.10. It is useful to study the behaviour of a face detection algorithm on whole images, rather than simple classification of whole images in "Face" and "Not Face" categories. In particular, it has been used in the literature to evaluate the precision of the CFF [50] and Viola-Jones [30].

We carried out our experiment as follows. We selected 500 images from the LFW_crop dataset [144] and 500 images from the Caltech101-background [142] dataset at random to build the training set, as in Section 3.1.3.1. Once again, we used it to train the

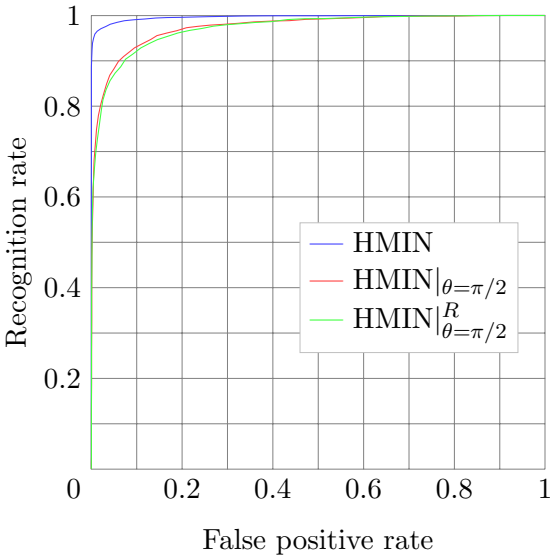


FIGURE 3.9: ROC curves of the HMIN classifiers on LFW_crop dataset. They show the recognition rate w.r.t the false positive rate: ideally, that curve would represent the function $\forall x \in (0, 1]\, f : x \to 0$ when $x = 0$, 1 otherwise. One can see a significant drop of performance when using $\text{HMIN}_{\theta=\pi/2}$ compared to HMIN – however using $\text{HMIN}^{R}_{\theta=\pi/2}$ does not significantly alter the accuracy. The drop of performance is to be put in perspective with the saving in terms of computational complexity.

RBF using the kerneling parameter $\mu = 2$. The images were all resized to 32×32, their histograms were equalized and we extracted features using $\text{HMIN}^{R}_{\theta=\pi/2}$; hence the feature vectors have 225 components.

After training, all images of the dataset were processed as follows. A pyramid is created from each images, meaning we built a set of the same image but with different sizes. Starting with the original size, the next image's width and height are 1.2 times smaller, which is 1.2 times bigger than the next, and so on until it is not possible to have an image bigger than 32×32. Then, 32×32 patches were cropped at all positions of all images of all sizes. Patches' histograms were equalized, and we extracted their $\text{HMIN}^{R}_{\theta=\pi/2}$ feature vectors which fed the RBF classifier.

We tested the accuracy of the classifications with several tolerance values, and accuracy were compared to the provided ground truth [143]. We use a definition of a *correctly detected face* close to what Garcia *et al.* proposed in [50]: we consider that a detection is valid if it contains both eyes and mouths of the face and the ROI's area is not bigger than 1.2 times the area of the square just wrapping the rectangle delimited by the eyes and mouths, i.e those square and rectangles share the same centroid and the width of the square is as long as the bigger dimension of the rectangle. For each face in the ground truth, we check that it was correctly detected using the aforementioned criterion – success counts as a "true positive", while failure counts as a "false negative". Then, for each region of the image that does *not* correspond to a *correctly detected face*, we check

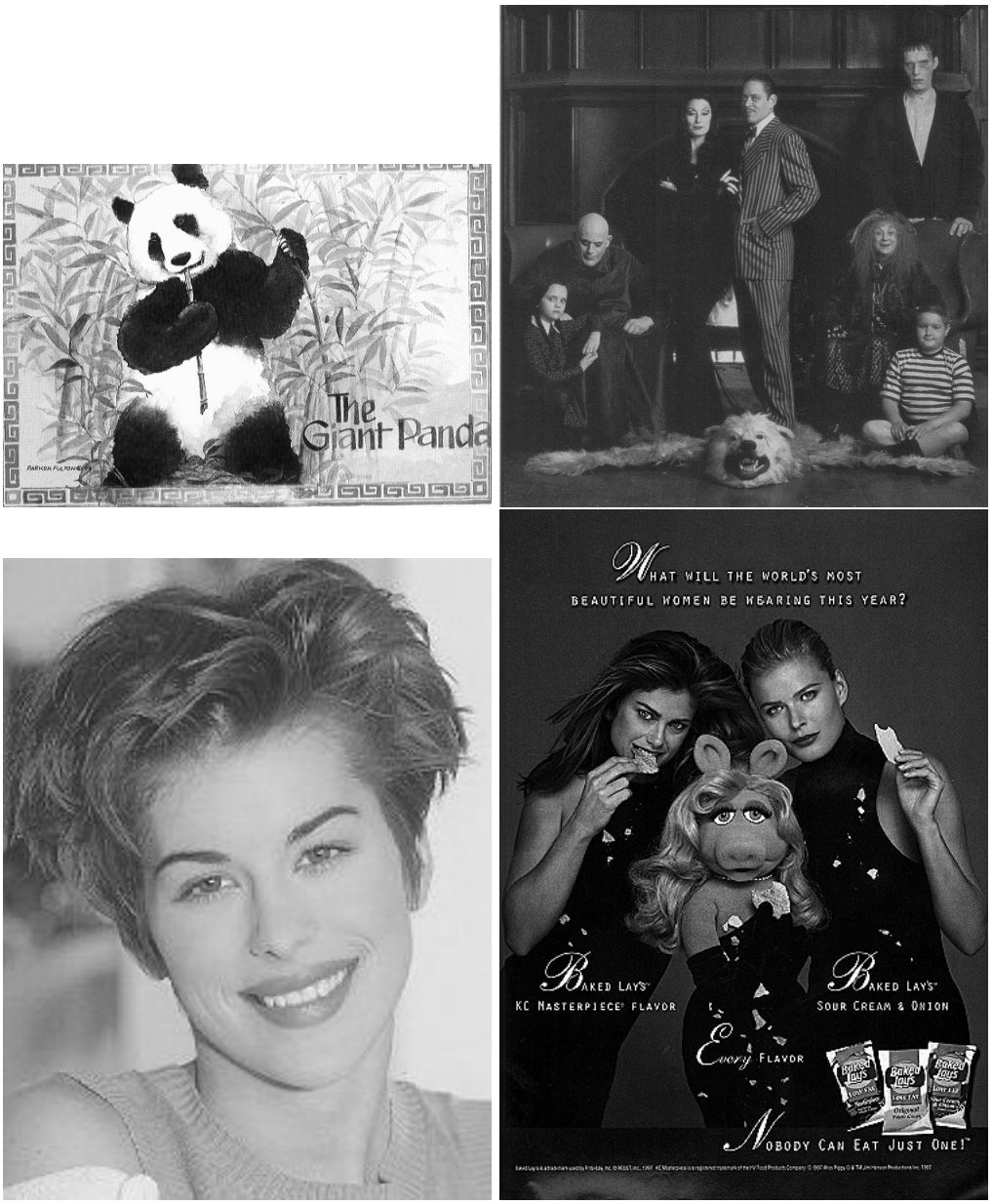


FIGURE 3.10: Samples from the CMU Face Images dataset.

if the system classified it as a "not-face" – in which case it counts as a "true negative" – or a face – in which case it counts as a "false positive". Some faces in CMU are too small to be detected by the system, and thus are not taken into account. The bigger image was removed because of its too high resolution. The resulting ROC curves are shown in Figure 3.11. Comparisons with CFF and VJ are sum-up in Table 3.2.

[1]We consider the case where the initial scale is 1 and $\Delta = 1$ – see [136] for more information.

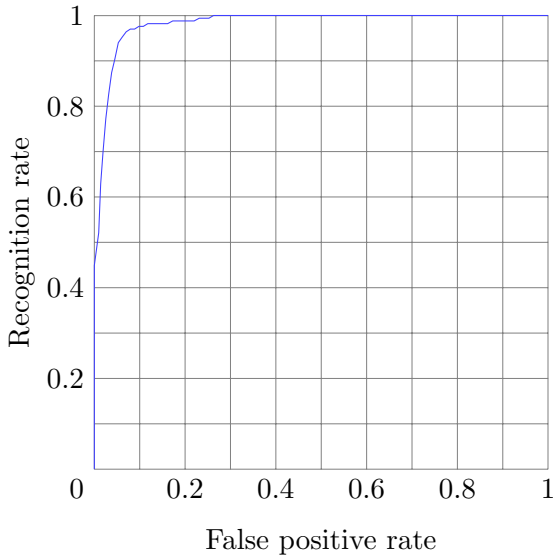


FIGURE 3.11: ROC curves obtained with $\text{HMIN}^R_{\theta=\pi/2}$ on CMU dataset. The chosen classifier is an RBF, and was trained with the features extracted from 500 faces from LFW_crop [141] dataset and 500 non-faces images cropped from images of the "background" class of the Caltech101 dataset [142]. For each image, a pyramid was produced in order to detect faces of various scales, were the dimensions of the images are successively reduced by a factor 1.2. A face was considered correctly detected if at least one ROI encompassing its eyes, nose and mouth was classified as "face", and if that ROI is not 20% bigger than the face according to the ground truth. Each non-face ROI that was classified as "Face" was counted as a false positive.

Framework	False positive rates (%)	Complexity (OP)		Memory print	Input size
		Scanning	Classification		
VJ	5.32×10^{-5} [136][1]	20.7 M	2.95 k	1.48 MB	24×24
CFF	5×10^{-5} [50]	50.7 M	129.5 k	64.54 MB	36×32
$\text{HMIN}^R_{\theta=\pi/2}$	4.5	26.1 M	82.9 k	1.2MB	32×32

TABLE 3.2: Complexity and accuracy of face detection frameworks. The false positive of the CFF and VJ frameworkw were drawn from the ROC curves of their respective papers [50, 136], and thus are approximate. All false positive rates are obtained with a 90% accuracy. The "Classification" column gives the complexity involved when computing a single patch of the size expected by the corresponding framework which is indicated in the "Input size" column. The "Frame" column indicates the complexity of the algorithm when scanning a 640×480 image. The complexities and memory prints shown here only take into account the feature extraction, and not the classification. It should be noted that in the case of the processing of an image pyramid, both CFF and HMIN would require a much higher amount of memory.

FIGURE 3.12: Example of frame from the "Olivier" dataset.

3.1.3.3 Test on Olivier dataset

In order to evaluate our system in more realistic scenarios, we created our own dataset specifically for that task. We acquired a video on a fixed camera of a person moving in front of a non-moving background, with his face looking at the camera – an example of a frame extracted from that video are presented in Figure 3.12. The training and evaluation procedure is the same as in Section 3.1.3.2: we trained an RBF classifier with features extracted with $\text{HMIN}^R_{\theta=\pi/2}$ from 500 images of faces from the LFW_crop dataset [141], and from 500 images cropped from images of the "background" class of the Caltech101 dataset [142]. We labeled the location of the face for each image by hand, so that the region takes both eyes and the mouth of the person, and nothing more, in order to be consistent with the CMU dataset [143]. Correct detections and false positives were evaluated using the same method as in Section 3.1.3.2: a face is considered as correctly detected if at least one ROI encompassing its eyes and mouth is classified as "face", and if that ROI is not more than 20% bigger than the face according to the ground truth. Each non-face ROI classified as a face is considered to be a false positive.

With that setting up, we obtained a 2.38% error rate for a detection rate of 79.72% – more detailed results are shown on Figure 3.13. Furthermore, we process the video frame by frame, without using any knowledge of the results from the previous images. For real-life scenarios, one could for instance get the detection not only on a single image, but on several, and validate the detection only if a point is detected on most of those frames as a face. That way, the number of false positives could be greatly reduced, while keeping a decent detection rate.

This Section was dedicated to our contribution in reducing the complexity of HMIN towards an embedded face detection application. Next Sections does a similar study for

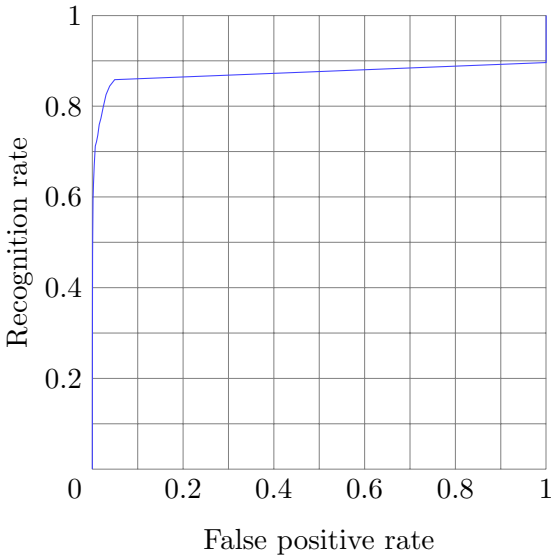


FIGURE 3.13: ROC curves obtained with $\text{HMIN}^{R}_{\theta=\pi/2}$ on "Olivier" dataset. As in Figure 3.11, the chosen classifier is an RBF, and was trained with the features extracted from 500 faces from LFW_crop [141] dataset and 500 non-faces images cropped from images of the "background" class of the Caltech101 dataset [142]. For each image, a pyramid was produced in order to detect faces of various scales, were the dimensions of the images are successively reduced by a factor 1.2. An image was considered correctly detected if at least one ROI encompassing its eyes, nose and mouth was classified as "face", and if that ROI is not 20% bigger than the face according to the ground truth. Each non-face ROI that was classified as "Face" was counted as a false positive.

a pedestrian detection application.

3.2 Feature selection for pedestrian detection

In this Section, we aim to propose a descriptor for pedestrian detection applications. The proposed descriptor is based on the same rational than in Section 3.1. Comparison in terms of computational requirements and accuracy shall be established between two of the most popular pedestrian detection algorithms.

3.2.1 Detecting pedestrians

With the arrival of autonomous vehicles, pedestrian detection rises as a very important issue nowadays. It is also vital in many security applications, for instance to detect intrusions in a forbidden zone. For this last scenario, one could think that a simple infrared camera could be sufficient – however such a device cannot determine by itself whether a hot object is really a human or an animal, which may be a problem in video-surveillance applications. It is then crucial to provide a method allowing to make that decision.

In this Section, we propose to use an algorithm similar to the one presented in Section 3.1.1, although this time it has been specifically optimized for the detection of pedestrian. One of the state of the art systems – which depends greatly on the considered dataset – is the work proposed by Sermanet *et al.* [145], in which they tuned a ConvNet for this specific task. However, as we shall see it requires lots of computational power, and we intend to produce a system needing as few resources as possible. Thus, we compare our system to another popular descriptor called HOG, which has proven efficient for this task. We shall now describe those two frameworks, then we shall study their computational requirements.

3.2.1.1 HOG

Histogram of Oriented Gradients (HOG) is a very popular descriptor, particularly well suited to pedestrian detection [36]. As its name suggests, it consists in computing approximations of local gradients in small neighborhoods of the image and use them to build histograms, which indicates the major orientations across small regions of the image. Its popularity comes from its very small algorithmic complexity and ease of implementation.

We focus here on the implementation given in [36], assuming RGB input images as for the face detection task presented in Section 3.1.1. The first step is to compute gradients at each position of the image. Each gradient then contributes by voting for the global orientation of its neighborhood. Normalization is then performed across an area of several of those histograms, thus providing the HOG descriptor that shall be used in the final classifier, typically SVM with linear kernels, that shall decides whether the image is of a person.

Gradients computation Using the same terminology as in [36], we are interested in the so called "unsigned" gradients, i.e we are not directly interested into the argument θ of the gradient, but rather $\theta \mod \pi$. Keeping that in mind, in order to compute the gradient at each location, we use an approximation implying convolution filters. All gradients are computed separately for each R, G and B channels – for each location, the only the gradient with the highest norm is kept.

Two feature maps H and V are produced from the input image respectively using the kernels $[-1, 0, 1]$ and $[-1, 0, 1]^{\mathrm{T}}$. At each location, values across the two feature maps at the same location may be seen as components of the 2D gradients, which we can use to compute their arguments and norms. Respectively denoting $G(x, y)$, $\phi_{[0,\pi]}((x, y))$ and $\|G(x, y)\|$ the gradient at location (x, y), its "unsigned argument" and its norm, and

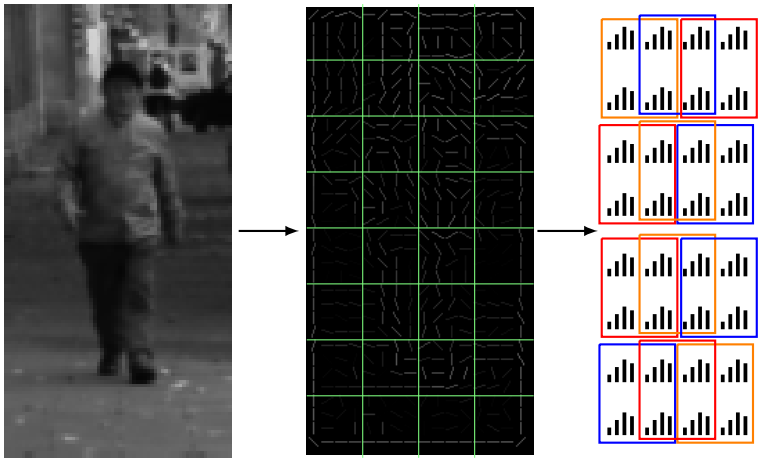


FIGURE 3.14: HOG descriptor computation. Gradients are computed for each location of the R, G and B channels, and for each location only the gradient with the highest norm is kept. The kept gradients are separated into *cells*, shown in green, and histograms of their orientations are computed for each cell. This produces a *histogram map*, which is divided in overlapping *blocks* a shown on the right. Normalization are performed for each block, which produces one feature vector per block. Those feature vectors are finally concatenated so as to produce the feature vector used for training and classification.

$H(x,y)$ and $V(x,y)$ the features from H and V feature maps at location (x,y), we have

$$\|G(x,y)\| = \sqrt{H(x,y)^2 + V(x,y)^2} \tag{3.49}$$

$$\phi_{[0,\pi]}(G(x,y)) = \arctan\left(\frac{V(x,y)}{H(x,y)}\right) \mod \pi \tag{3.50}$$

The result of that process is shown in Figure 3.14. It is important to note here that the convolutions are performed so that the output feature maps have the same width and height as the input image. This may be ensured by cropping images slightly bigger than actually needed, or by padding the image with 1 pixel at each side of its side with 0's or replicating its border.

Binning Now that we have the information we need about the gradients, i.e their norms and arguments, we use them to perform the non-linearity proposed in this framework. The image is divided in so-called *cells*, i.e non-overlapping regions of $N_c \times N_c$ pixels, as illustrated in Figure 3.14. For each *cell*, we compute an histogram as follows. The half-circle of unsigned angles is evenly divided into B *bins*. The center c_i of the i-th bin is given by the centroid of the bin's boundaries, as shown in Figure 3.15. Each gradient in the cell votes for the two bins with the centers closest to its argument. Calling those bins c_l and c_h, the weights of its votes w_l and w_h depend on the difference

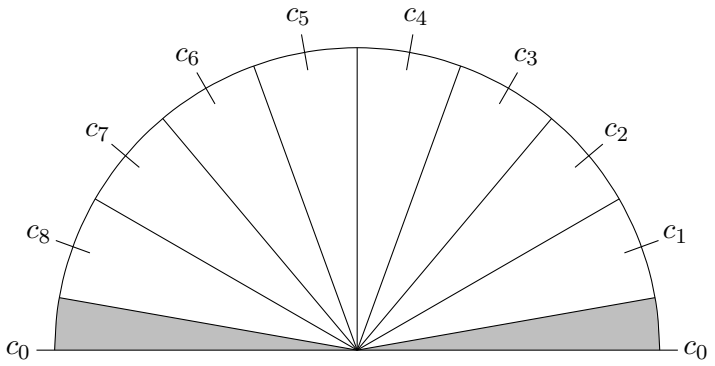


FIGURE 3.15: Binning of the half-circle of unsigned angles with $N_b = 9$. The regions in gray correspond to the same bin.

between its argument and the bin center, and on its norm:

$$w_h = |G(x,y)| \frac{\phi(G(x,y)) - c_l}{c_h - c_l} \tag{3.51}$$

$$w_l = |G(x,y)| \frac{\phi(G(x,y)) - c_h}{c_h - c_l} \tag{3.52}$$

We end up having a histogram per *cell*. Assuming the input image is of size $W \times H$ and that N_c both divide W and H, we have a total of $WH/{N_c}^2$ histograms. We associate each histogram to its corresponding cell to build a so called *histogram map*.

Local normalization The last step provides some invariance to luminosity among histograms. The *histogram map* is divided into overlapping *blocks*, each having 2×2 histograms. The stride between two overlapping blocks is 1 so that the whole histogram map is covered. All the bins' values of those histograms form a vector $v(x_h, y_h)$ having BN_b^2 components where (x_h, y_h) is the location of the top-left corner's of the *block* in the *histogram map* frame coordinate, and we compute its normalized vector $\overline{v}(x_h, y_h) = \left(\overline{v_1}(x_h, y_h), \overline{v_2}(x_h, y_h), \ldots, \overline{v_{{N_b}^2}}(x_h, y_h)\right)$ using the so called *L2-norm* [36] normalization:

$$\forall i \in \left\{1, \ldots, B{N_b}^2\right\} \quad \overline{v_i}(x_h, y_h) = \min\left(\frac{v_i(x_h, y_h)}{\sqrt{\|v(x_h, y_h)\|^2 + \epsilon^2}}, 0.2\right) \tag{3.53}$$

where ϵ is a small value avoiding divisions by 0.

Thus we obtain a set of vectors $\overline{v}(x_h, y_h)$, which are finally concatenated in order to form the feature vector fed in a SVM classifier.

Complexity analysis Let's evaluate the complexity of extracting HOG features from an $W \times H$ image. As we saw, the first step of the extraction is the convolutions, that require of $6WH$ operations per channel, followed by the computation of their squared norms, which requires $3WH$ operations per channel; thus at this point we need $3(3 + 6)WH = 27WH$ operations. Afterward, we need to compute the maximum values across the three channels for each location, thus leading to $2WH$ more operations. Finally, we must compute the gradients, which we assume involves one operation for the division, one operation for the arc-tangent and one for the modulus operation; hence $3WH$ more operations. Thus, the total amount of operations at this stage is given by

$$C_{\text{HOG}_{\text{grad}}} = 32WH \tag{3.54}$$

Next, we perform the binning. We assume that finding the lower and higher bins takes two operations: one for finding the lower bin, and another one to store the index of the higher bin. From Equation 3.51, we see that computing w_h takes one subtraction and one division, assuming $c_h - c_l$ is pre-computed, to which we add one operation for the multiplication with $|G(x, y)|$, thus totaling 3 operations. The same goes for the computation of w_l. Finally, w_h and w_l are both accumulated to the corresponding bins, requiring both one more operations. This done at each location of the feature maps, thus this stage needs a total of operations of

$$C_{\text{HOG}_{\text{hist}}} = 8WH. \tag{3.55}$$

Still neglecting the complexity involved by the SVM classifier, the only remaining step is the normalization. It consists in normalizing vectors concatenating the 2×2 histograms of 9 bins, which represent vectors of 36 components, at all possible location of the histogram map where we have valid data. The histogram map being $W_h \times H_h$, we perform that normalization at N_p positions, with

$$N_p = (W_h - 1) \times (H_h - 1) \tag{3.56}$$

positions, with

$$W_h = \frac{W}{8}, \tag{3.57}$$

$$H_h = \frac{H}{8}. \tag{3.58}$$

Each histogram having 9 bins and since they each normalization needs 4 of them, the vectors to normalize have 36 components. From Equation 3.53, we see that we need to compute the square of a Euclidean distance, a sum, a square root, followed by a division

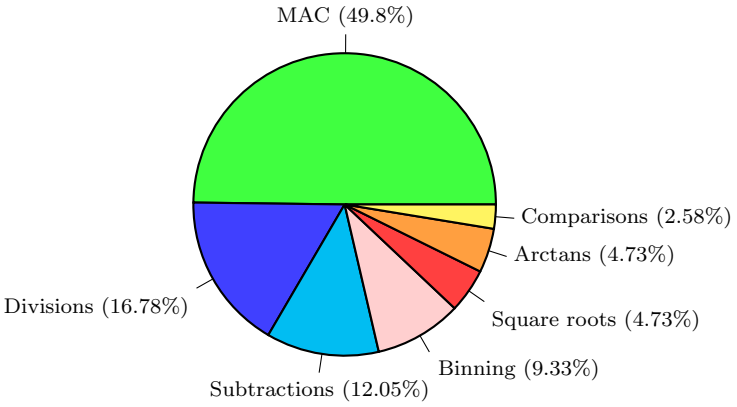


FIGURE 3.16: Complexity repartition of HOG features extraction.

of each component of the vector by a scalar, and finally a comparison. Since the sum and the square root may be considered to take a single operation, which is very small compared to the total, we chose to neglect it to make the calculation more tractable. The Euclidean distance itself requires one subtraction followed by a MAC operation per component. Thus, each vector normalization needs 2×36 operations for the Euclidean distance, 36 divisions and 36 comparison operations, which to a total of $4 \times 36 = 144$ operations per position. Equations 3.56 to 3.58 then give

$$C_{\text{HOG}_{\text{NORM}}} = 4 \times 46 \left(\frac{W}{8} - 1\right)\left(\frac{H}{8} - 1\right) \tag{3.59}$$

$$= 144 \left(\frac{WH}{64} - \frac{W+H}{8} + 1\right) \tag{3.60}$$

$$= 2.25WH - 18\,(W+H) + 144. \tag{3.61}$$

And thus, combining Equations 3.54 to 3.61, we get

$$C_{\text{HOG}} = C_{\text{HOG}_{\text{grad}}} + C_{\text{HOG}_{\text{hist}}} + C_{\text{HOG}_{\text{norm}}} \tag{3.62}$$

$$C_{\text{HOG}} = 42.25WH - 8\,(W+H) + 144 \tag{3.63}$$

Thus, extracting features from a 64×128 image as suggested in [36] takes 344.7 kOP. When scanning an image to locate pedestrians, we may use the same method as usual [48, 50]. Using Equation 3.63 on a 640×480 image, we get a complexity of **12.96 MOP**. Repartitions of the computational efforts are presented in Figure 3.16.

Memory print Let's now evaluate the memory print required by the extraction of HOG features for a 640×480 input image. When computing the gradients, the first step consists in performing 2 feature maps from convolutions, of the same size of the input image. We consider here each feature of the feature maps shall be coded as 16

bits integers, hence we need $2 \times 2 \times 640 \times 480 = 1.23 \times 10^6$ bytes at this stage. Then, the modulus and arguments of the gradients are computed at each feature location. We assume here that that data shall be stored using single precision floating point scheme; hence 32 bits per value, and then we need 2.45 MB. As for the histograms, since there is no overlaps between cells, they may be evaluated in-place – hence, they do not bring more memory requirement. Finally comes the memory needed by the normalization stage; assuming we neglect the border effect, one normalized vector is computed at each cell location, which correspond to 8×8 areas in the original image. Hence, 4800 normalized vectors are computed, each having 36 component, which leads to 691.2 kB. Thus, the memory print of the HOG framework is 4.37 MB.

We presented an analysed the HOG algorithm for pedestrian detection. In the next Section, we describe a particular architecture of a ConvNet optimized for that same task.

3.2.1.2 ConvNet

As for many other applications, ConvNet have proven very efficient for pedestrian detection. Sermanet *et al.* proposed in [145] a ConvNet specifically designed for that purpose.

Presentation We now review the architecture of that system, using the same notations as in Section 3.1.1.2. First of all, we assume images use the Y'UV representation. In this representation, the Y channel represents the *luma*, i.e the luminosity, while the U and V channels represent coordinates of a color in a 2D space. The Y channel is processed separately from the UV channels in the ConvNet.

The Y channel first goes through the C_Y1 convolution stage which consists in 32 kernels, all 7×7, followed by an absolute-value rectification – i.e we apply a point-wise absolute value function on all output feature maps [146] – followed by a local constrast normalization which is performed as followed [145]:

$$v_i = m_i - m_i \star w \tag{3.64}$$

$$\sigma = \sqrt{\sum_{i=1}^{N} w \star {v_i}^2} \tag{3.65}$$

$$y_i = \frac{v_i}{\max(c, \sigma)} \tag{3.66}$$

where m_i is the i-th un-normalized feature map, $\star$ denotes the convolution operator, w is a Gaussian blur 9×9 convolution kernel with normalized weights, N is the number of

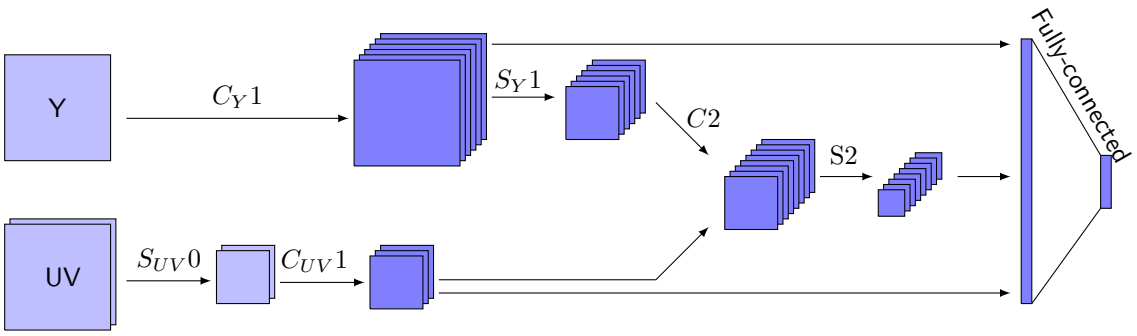


FIGURE 3.17: ConvNet for pedestrian detection [145]. Input image is assumed to be represented in Y'UV space. The Y channel feed the C_Y1 convolution layer, the resulting feature maps of which are sub-sampled in S_Y1. In parallel, the UV channels are subsampled by the $S_{UV}0$ layer, and the results feed the $C_{UV}1$ convolution layer. The $C_{UV}1$ and S_Y1 feature maps are concatenated and feed the $C2$ convolution layer. The $C2$ feature maps are then subsampled by $S2$. Finally, all output features from $C2$ and $C_{UV}1$ are serialized and used as inputs of a fully-connected layer for classification.

feature maps to normalize and c is a constant. Finally, a 3×3 average down-sampling – meaning that each feature of the sub-sampled feature map is the result of a 3×3 average-pooling.

The U and V channels are subsampled by 3 using the same averaging scheme in the $S_{UV}0$ layer. Those sub-sampled feature maps feed the $C_{UV}1$ stage which consists in 12 5×5 convolution kernels which output 6 feature maps, followed by an absolute-value rectification and a local contrast normalization.

All feature maps from C_Y1 and $C_{UV}1$ are then concatenated, thus providing a total of 38 feature maps which feed the $C2$ stage. It consists in 2040 9×9 convolution filters that produce 68 feature maps. Absolute-value rectification is applied to them, and they are subsampled by 2×2 in the $S2$ layer. All features from the C_Y1, $C_{UV}1$ and $C2$ are concatenated to form the feature vector to be classified, which is performed with a classical linear classifier. That architecture is sum-up in Figure 3.17.

Complexity analysis Let's now evaluate the amount of operations needed for a $W \times H$ Y'UV image to be processed by that ConvNet. Denoting C_X the complexity

involved in layer X and along the lines of the calculus done in Section 3.1.1.2, we have

$$C_{C_Y1} = 32 \times 7 \times 7 \times (W-6)(H-6) \tag{3.67}$$

$$C_{S_Y1} = 32 \times 9 \times \left\lfloor \frac{W-6}{3} \right\rfloor \left\lfloor \frac{H-6}{3} \right\rfloor \tag{3.68}$$

$$C_{S_{UV}0} = 2 \times 9 \times \left\lfloor \frac{W}{3} \right\rfloor \left\lfloor \frac{H}{3} \right\rfloor \tag{3.69}$$

$$C_{C2} = 2040 \times 9 \times 9 \times 2 \times (W_{S_{UV}0} - 8)(H_{S_{UV}0} - 8) \tag{3.70}$$

$$C_{S2} = 68 \times 2 \times 2 \left\lfloor \frac{W_{C2}}{2} \right\rfloor \left\lfloor \frac{H_{C2}}{2} \right\rfloor \tag{3.71}$$

where W_X and H_X respectively denote the width and height of the X feature maps. The $C_{UV}1$ layer has full connection between its input and output feature maps. Thus, denoting N_I and N_O respectively the number of input and output feature maps, a total of $N_I N_O$ convolutions are performed. Inside this layer, this produces $N_I N_O$ feature maps, which are sum feature-wise so as to produce the N_O output feature maps. This leads to

$$C_{UV}1 = 2 \times 6 \times 6 \times (W_{S_{UV}0} - 4)(H_{S_{UV}0} - 4)\,. \tag{3.72}$$

We shall now evaluate the complexity involved by the absolute value rectifications which are performed on the C_Y1 and $C2$ feature maps. It needs one operation per feature, thus denoting $C(A_X)$ the complexity involved by those operations on feature map X we have

$$C_{A_{C_Y1}} = 32 W_{C_Y1} H_{C_Y1} \tag{3.73}$$

$$C_{A_{C_{UV}1}} = 6 W_{C_{UV}1} H_{C_{UV}1} \tag{3.74}$$

$$C_{A_{C2}} = 68 W_{C2} H_{C2}. \tag{3.75}$$

Finally, we evaluate the complexity brought by the local contrast normalizations. From Equations 3.64, 3.65 and 3.66, we see that the first step consists in a convolution by a 9×9 kernel G followed by a pixel-wise subtraction between two feature maps. Assuming the input feature map is $w \times h$ and that the convolution is performed so that the output feature map is the same size as the input feature map, the required amount of operations at this step is given by

$$C_{N_1}(w,h) = 2 \times 9 \times 9 \times wh = 162wh. \tag{3.76}$$

The second step involves squaring up each feature of the wh output feature maps, which implies wh operations. The result is again convolved with G, implying $81wh$ operations, and the resulting feature are sum feature-wise across N feature maps, implying nwh

sums. Finally, we produce a "normalization map" by taking the square root of all features, which involves wh operations assuming a square root takes only one operation. Hence:

$$C_{N_2}(w,h,n) = (83+n)\,wh \tag{3.77}$$

The final normalization step consists in computing, for each feature of the normalization map, the maximum value between that feature and the constant c, which leads to wh operations, and perform feature-wise divisions between the N maps computed in Equation 3.64 and those maximums, which leads to nwh operations. Thus we have

$$C_{N_3}(w,h,n) = (1+n)\,wh, \tag{3.78}$$

and the complexity brought by a local contrast normalization on n $w \times h$ feature maps is given by

$$C_N(w,h,n) = (246+2n)\,wh. \tag{3.79}$$

The overall complexity is given by

$$\begin{aligned} C_{\text{ConvNet}} &= C_{C_Y1} + C_{S_Y1} + C_{S_{UV}0} \\ &\quad + C_{C2} + C_{S2} \\ &\quad + C_{A_{C_Y1}} + C_{A_{C_{UV}1}} + C_{A_{C2}} \\ &\quad + C_N(W_{C_Y1}, H_{C_Y1}, 32) + C_N(W_{C_{UV}1}, H_{C_{UV}2}, 6) + C_N(W_{C1}, H_{C2}, 68) \end{aligned} \tag{3.80}$$

which leads to

$$\begin{aligned} C_{\text{ConvNet}} &= 1568 W_{C_Y1} H_{C_Y1} + 288 W_1 H_1 \\ &\quad + 18 W_{S_{UV}0} H_{S_{UV}0} + 330480 W_{C2} H_{C2} \\ &\quad + 272 W_{S2} H_{S2} + 24 W_1 H_1 \\ &\quad + 342 W_{C_Y1} H_{C_Y1} + 264 W_{C_{UV}1} H_{C_{UV}1} + 450 W_{C2} H_{C2} \end{aligned} \tag{3.81}$$

with

$$W_{C_Y1} = W - 6 \tag{3.82}$$

$$H_{C_Y1} = H - 6 \tag{3.83}$$

$$W_{S_{UV}0} = \left\lfloor \frac{W}{3} \right\rfloor \tag{3.84}$$

$$H_{S_{UV}0} = \left\lfloor \frac{H}{3} \right\rfloor \tag{3.85}$$

$$W_1 = W_{S_Y1} = W_{C_{UV}1} = \left\lfloor \frac{W_{C_Y1}}{3} \right\rfloor = W_{C_{UV}0} - 4 \tag{3.86}$$

$$H_1 = W_{S_Y1} = H_{C_{UV}1} = \left\lfloor \frac{H_{C_Y1}}{3} \right\rfloor = H_{C_{UV}0} - 4 \tag{3.87}$$

$$W_{C2} = W_1 - 8 \tag{3.88}$$

$$H_{C2} = H_1 - 8 \tag{3.89}$$

$$W_{S2} = \left\lfloor \frac{W_{C2}}{2} \right\rfloor \tag{3.90}$$

$$H_{S2} = \left\lfloor \frac{H_{C2}}{2} \right\rfloor \tag{3.91}$$

Let's evaluate this expression as a function of the width W and height h of the input image. In order to make it more tractable, we approximate it by neglecting the floor operators $\lfloor\rfloor$. Reusing Equation 3.81 to 3.91 we have

$$C_{\text{ConvNet}}(W, H) \approx 38.8 \times 10^3 WH - 1.12 \times 10^6 (W + H) + 33.2 \times 10^6 \tag{3.92}$$

It should be noted that we again neglected the classification stage. Considering input images are 78×126, we have $C_{\text{ConvNet}} \approx 484.84$ MOP.

Applying Equation 3.92 to the case where we process a 640×480, we have 11 GOP. From the previous analysis, we see that lots of MAC are computed at almost all stage, including the average downsampling ones. This is largely due to the C2 layer, with its high amount of convolution filters. It is then clear that optimization efforts should be directed towards the computation of MACs.

Memory print Let's now evaluate the memory print of that framework when processing a 640×480 input image. The C_{Y1} layer produces 32 634×474 feature maps, in which we assume the features are coded using 32-bits floating point precision, which needs 38.47 MB. In order to simplify our study, we then assume that the subsampling and normalization operations are performed in-place, and hence do not bring more need in memory. The S_{Y1} layer produces 2 213×160 feature maps, hence needing 272.64 kB.

Finally the S2 layer produces 2040 102×76 feature maps, which using 32 bits floating point precision would require 63.26 MB.

3.2.2 HMAX optimizations for pedestrian detection

We propose optimizations along the lines of what was explained in Section 3.1.2. When we were looking for faces, we hand-crafted the convolution kernel so that it responded best to horizontal features, in order to extract eyes and mouths for instance. However, in the case of pedestrians it intuitively seems more satisfactory to detect vertical features. Thus, we propose to keep the same kernel as represented in Figure 3.7, but flipped by 90°. As in Section 3.1.2, we have two descriptors: $\text{HMIN}_{\theta=0}$ and $\text{HMIN}^R_{\theta=0}$. For consistency reasons with what was done for faces in Section 3.1.2 and with the HOG [36] and ConvNet [145] algorithms, $\text{HMIN}_{\theta=0}$ expects 64×128 input images and consists in a single 37×37 convolution kernel. As for $\text{HMIN}^R_{\theta=0}$, it expects 16×32 inputs and consists in a 9×9 convolution kernel.

3.2.3 Experiments

In order to test our optimizations, we used the INRIA pedestrian dataset, originally proposed to evaluate the performances of the HOG algorithm [36]. That dataset is divided in two subsets: a training set and a testing set. Hence, we simply trained the system described in Section 3.2.2 on the training set and evaluated it on the testing set. Results are shown in Figure 3.18, which is a ROC curve produced as done for faces in Section 3.1.3.1. All images were resized to 16×32 before process. Comparisons with HOG and ConvNet features are shown in Table 3.3.

In this Section, we proposed and evaluated optimizations for the so-called HMIN descriptor applied to pedestrian detection. Next Section is dedicated to a discussion about the results that we obtained both here, and in the previous Section which was related to face detection.

3.3 Discussion

Let's now discuss the results obtained in the two previous Sections, where we described a feature extraction framework and compared its performance, both in terms of accuracy and complexity, against major algorithms.

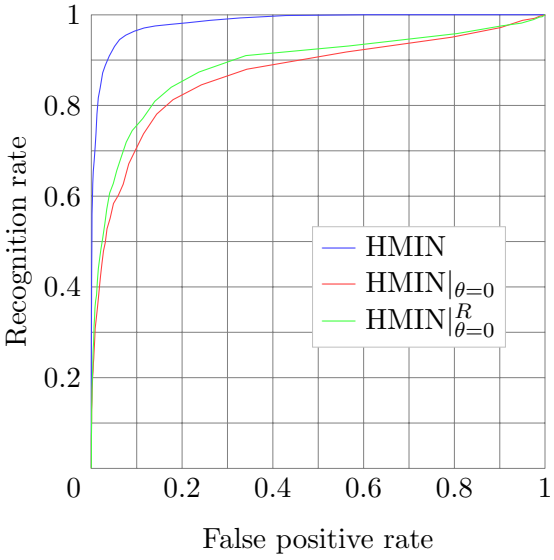


FIGURE 3.18: ROC curves of the HMIN classifiers on the INRIA pedestrian dataset. The drop of performance is more important here than it was for faces, as shown on Figure 3.9. However, the gain in complexity is as significant as in Section 3.1.2.

Framework	False positive rate (%)	Complexity (OP)		Memory print	Input size
		Scanning	Classification		
HOG	0.02 [36]	12.96 M	344.7 k	4.37MB	64×128
ConvNet	See caption	484.84 M	11 G	63.26MB	78×126
$\text{HMIN}^{R}_{\theta=0}$	30%	13.05 M	41.45 k	1.2 MB	32×16

TABLE 3.3: Complexity and accuracy of human detection frameworks. The false positive rate of the HOG has been drawn from the DET curve shown in [36], and thus is approximate. The false positive rates presented here correspond to a 90% detection rate. As in Table 3.2, the "Classification" column gives the complexity involved when computing a single patch of the size expected by the corresponding framework which is indicated in the "Input size" column. The "Frame" column indicates the complexity of the algorithm when applied to a 640×480 image. Furthermore, the complexities involved by HMIN are computed from Equation 3.46, with the input size shown in the column on the right. Finally the result of the ConvNet may not be shown here as their strategy for evaluating it is different from what was done in [36] – using the evaluation protocol detailed in [145], HOG produces approximately three times as many false positives as ConvNet. Furthermore, the miss rate of the HOG was determined on a scene-scanning task, while we evaluated our framework on a simpler classification task. Thus, comparisons of the accuracy of those frameworks are difficult, although the preliminary results presented here show a clear disadvantage in using $\text{HMIN}^{R}_{\theta=0}$. Finally, the complexities and memory prints shown here only take into account the feature extraction, and not the classification. It should also be noted that both are evaluated without image pyramid, and that in that case they would be much higher than evaluated here.

Results of our framework are sum up in Table 3.2 for face detection applications and in Table 3.3 for pedestrian detection application. First of all, we see from the ROC curves shown in Figures 3.11 and 3.18 that the accuracy of our framework is significantly bigger for face detection tasks than for pedestrian detection task – although comparing performances on two different tasks is dangerous, those results seem to indicate that our framework would operate much better in the first case. However, the raw accuracy is significantly lower than those of the other frameworks presented here, be it for face or human detection. This is probably due to the fact that our frameworks $\text{HMIN}^R_{\theta=x}$ [2] produce features that are much simpler than those of the other frameworks – indeed, the feature vector for a 32×32 input image has only 225 components. Among all other frameworks, the only other that may be considered better to that respect is Viola-Jones, where on average only 8 features are computed, although in the worst case that amount rises dramatically to 6060.

Nevertheless, Viola-Jones and the HOG algorithms are both slightly less complex than $\text{HMIN}_\theta = x^R$. There is also a consequent literature about their implementations on hardware [147–163]. In particular, the main difficulties of the HOG algorithm for hardware implementations, i.e the highly non-linear computations of the arc-tangents, divisions and square roots, have been addressed in [149]. As for the CFF, it was also optimized and successfully implemented on hardware devices [100] and on embedded processors [164, 165].

However, one can expect $\text{HMIN}^R_{\theta=x}$ to be implemented easily on FPGA, with really low resource utilization – that aspect shall be tested in future development. Furthermore, the only framework that beats HMIN in terms of memory-print is Viola-Jones – that aspect is crucial when porting an algorithm on an embedded systems, especially in industrial use cases where constraints may be really high in that respect. Furthermore, while $\text{HMIN}^R_{\theta=x}$ may not seem as attractive as the other frameworks presented here, it has a very interesting advantage: it is generic. Indeed, both ConvNet implementations presented in this Chapter were specifically designed for a particular task: face detection or pedestrian detection. As for Viola-Jones, it may be used for tasks other than face detection as was done for instance for pedestrian detection [166] – however, a different task might need different Haar-like features, which would be implemented differently than the simple ones presented in Section 3.1.1.1. In terms of hardware implementation, that difference would almost certainly mean code modifications, while with $\text{HMIN}^R_{\theta=x}$ one would simply need to change the weights of the convolution kernel. Concerning the HOG, it should be as generic as HMIN – however it suffers from a much greater memory print.

[2] $\text{HMIN}^R_{\theta=x}$ refers to both $\text{HMIN}^R_{\theta=0}$ and $\text{HMIN}^R_{\theta=\pi/2}$.

Finally, researchers have also proposed other optimization schemes for HMIN [46, 167] – future research shall focus on comparing our work with the gained one can expect with their solutions, as well as use a common evaluation scheme for the comparison of $\text{HMIN}^R_{\theta=0}$ with other pedestrian detection algorithms.

3.4 Conclusion

In this Chapter, we presented our contribution concerning the optimization of a feature extraction framework. The original framework is based on an algorithm called HMAX, which is a model of the early stages of the image processing by the mamal brain. It consists in 4 scales, called S1, C1, S2 and C2 – however, in the use case scenarions presented here the S2 and C2 layers do not provide much more precision, but are by far the most costly in terms of algorithm complexity. We thus chose to keep only the S1 and C1 layers, respectively consisting in a convolution filter bank and max-pooling operations. We explored how the algorithm behaved when diminuing its complexity, by reducing the number and sizes of the linear and max-pooling filters, by estimating where the most relevant information is located.

We replaced the initial 64 filters in the S1 layer with only one, the size of which is 9×9. It expects 32×32 grayscale images as inputs. The nature of the filter depends on the use case: for faces, we found that most saliencies lie in the eyes and mouth of the face, thus we chose a filter responding to horizontal features. As for the use case of human detection, we assume that pedestrians are standing up, which intuitively made us use a filter responding to vertical features. In both cases, we compared the results with standard algorithms having reasonable complexities. Optimizing out the HMIN descriptor provoked a drop in accuracy of 5.73 points on the face detection task on the CMU dataset, and 21.91 points on the pedestrian detection task when keeping a false positive rate of 10%. However, that drop of performance is to be put in perspective with the gain in complexity: after optimizations, the descriptor is 429.12 less complex to evaluate. In spite of everything, that method does not provide results as good as other algorithms with comparable complexities, e.g Viola-Jones for face detection – as for pedestrian detection, we need to perform complementary tests with common metrics for the comparison of that system with the state of the art, but the results presented here tend to show that that algorithm is not well suited for this task. However, we claim that our algorithm provides a low memory print and is more generic than the other frameworks, which make it implementable on hardware with fewer resources, and should be easy to adapt for new tasks: only the weights of the convolution kernel are to be changed.

This Chapter was dedicated to the proposition of optimizations for a descriptor. Next Chapter will present another type of optimizations, not based on the architecture of the algorithm, but on the encoding of the data, with implementation on a dedicated hardware. As we shall see, those optimizations are much more efficient and promising, and may easily be applied to other algorithms.

Chapter 4

Hardware implementation

This chapter addresses the second question stated in Chapter 2, about the optimization of the HMAX framework with the aim of implementing it on a dedicated hardware platform. We begin by exposing the optimizations that we used, coming both from our own work and from the literature. In particular, we show that the combination of all those optimizations does not bring a severe drop in accuracy. We then implement our optimized HMAX on an Artix-7 FPGA, as naively as possible, and we compare our results with those of the state of the art implementation. While our implementation achieves a significantly lower throughput, we shall see that it uses much less hardware resources. Furthermore, our optimizations are fully compatible with those of the state of the art, and future implementations may profit from both contributions.

4.1 Algorithm-Architecture Matching for HMAX

In the case of embedded systems, having an implemented model in a high-level language such as Matlab is not enough. Even an implementation using the C language may not meet the particular constraints that are found in critical systems, in terms of power consumption, algorithmic complexity and memory print. This is particularly true in the case of HMAX, where the S2 layer in particular may take several seconds to be computed on a CPU. Furthermore, GPU implementations are most of the time not an option, as GPUs often have a power consumption in the order of magnitude of 10 W. In the fields of embedded systems, we look for systems consuming about 10 to 100 mW. This may be achieve thanks to FPGAs, as was done in the past [91–96, 98, 99]. This Chapter proposes a detailed review of one of those implementations; the other ones are either based on architecture with multiple high-end FPGAs or focus on accelerating a

part of the framework only, thus they are hardly comparable with what we aim to do here.

Orchard *et al.* proposed in [99] a complete hardware implementation of HMAX on a single Virtex-6 ML605 FPGA. To achieve this, the authors proposed optimizations on their own, which concern mostly the way the data is organized and not so much the encoding and the precision degradation – indeed, the data coming out of S1 and carried throughout the processing layers is coded on 16 bits.

We shall now review the main components of their implementation, e.g. the four modules implementing the behaviours of S1, C1, S2 and C2. The layers are pipelined, so they may process streamed data. As for the classification stage, it is not directly implemented on the FPGA and should be taken care of on a host computer. The results of that implementation are presented afterwards.

4.1.1 Description

4.1.1.1 S1

First of all, the authors showed how all filters in S1 may be decomposed as separable filters, or sums of seperable filters. Indeed, if we consider the "vertical" Gabor filters in S1, i.e. we have $\theta = \pi/2$, Equations 2.8 and 2.9 lead to [99]

$$G(x,y)\big|_{\theta=\pi/2} = \exp\left(-\frac{x^2+\gamma^2y^2}{2\sigma^2}\right)\cos\left(\frac{2\pi}{\lambda}x\right) \tag{4.1}$$

$$= \exp\left(-\frac{x^2}{2\sigma^2}\right)\cos\left(\frac{2\pi}{\lambda}x\right) \times \exp\left(-\frac{\gamma^2y^2}{2\sigma^2}\right) \tag{4.2}$$

$$= H(x)\,V(y) \tag{4.3}$$

$$H(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)\cos\left(\frac{2\pi}{\lambda}x\right) \tag{4.4}$$

$$V(y) = \exp\left(-\frac{\gamma^2y^2}{2\sigma^2}\right), \tag{4.5}$$

thus, by denoting $*$ the convolution operator and I the input image:

$$I * G\big|_{\theta=0} = I *_c V *_r H \tag{4.6}$$

where $A *_r B$ denotes separated convolutions on rows of 2D data A by 1D kernel B, and $A *_c B$ denotes column-wise convolutions of A by B. Using the same notations:

$$G(x,y)\big|_{\theta=0} = G(y,x)\big|_{\theta=\pi/2} \tag{4.7}$$

and then

$$I * G\,\big|_{\theta=\pi/2} = I *_c H *_r V \tag{4.8}$$

Let's now focus on the filters having "diagonal" shapes. As shown in [99] and following the same principles as before, we may write

$$I * G\,\big|_{\theta=\pi/4} = I *_c H *_r H + I *_c U *_r U \tag{4.9}$$

$$I * G\,\big|_{\theta=3\pi/4} = I *_c H *_r H - I *_c U *_r U \tag{4.10}$$

with

$$U(x,y) = \exp\left(-\frac{x^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda}x\right). \tag{4.11}$$

The benefits in using separable filters are twofolds. First of all, the memory prints of those filters are much smaller than their unoptimized counterparts. Indeed, storing a $N\times N$ filter in a naive way requires storing N^2 words, while their separated versions would require the storage of $2N$ words for $G|_{\theta=0}$ and $G|_{\theta=\pi/2}$, and $3N$ words for $G|_{\theta=\pi/4}$ and $G|_{\theta=3\pi/4}$. The other benefit is related to the algorithmic complexity. Indeed, performing the convolution of a $W_I \times H_I$ image by a $W_K \times H_K$ kernel has an $O(W_I H_I W_K H_K)$, while for separable filters it goes down to $O(W_I W_K + H_I H_K)$. According to [99], doing so reduces the complexity from 36,146MAC operations to 2816MAC.

In order to provide some invariance to luminosity, Orchard *et al.* also use a normalization scheme called l^2. Mathematically, computing that norm consists in taking square root of the sum of the pixels. Gabor filters where thus normalized so that their l^2 norms equal $2^{16}-1$, and so that their means are null.

4.1.1.2 C1

Let's consider a C1 unit with a $2\Delta \times 2\Delta$ receptive field. The max-pooling operations are performed as follows: first, maximums are computed in $\Delta \times \Delta$ neighborhoods, producing an intermediate feature map M_t. Second, the output of the C1 unit are obtained by pooling over 2×2 pooling windows from M_t with a overlap of 1. This elegant method allows to avoid the storage of values that would have been discarded any way, as the data is processed here as it is provided by S1, in a pipelined manner.

4.1.1.3 S2

In the original model, it is recommended to use 1000 pre-learnt patches in S2. However, the authors used themselves 1280 of them – 320 per classes – as it was the maximum

Resource	Used	Available	Utilization(%)
DSP	717	768	93
BRAM	373	416	89
Flip-flops	66196	301440	21
Look-up tables	60872	150720	40

TABLE 4.1: Hardware resources utilized by Orchard's implementation [99].

amount that could fit on their device. At each location, pattern-matching are multiplexed by size, i.e first all $4 \times 4 \times 4$ in parallel, then $8 \times 8 \times 4$, then $12 \times 12 \times 4$ and finally $16 \times 16 \times 4$. Responses are computed for two different orientations in parallel, this results in a total of $320 \times 2 = 640$ MAC operations to be performed in parallel at each clock cycle. Thus, this requires 640 multipliers, and 640 coefficients to be read at each clock cycle. As for the precision, each feature is coded on 16 bits to fit.

4.1.1.4 C2

Due to the simplicity of C2 in the original model, there is not much room for optimizations or implementation tricks here. Orchard *et al.*'s implementation simply gets the 320 results from S2 in parallel and use them to perform the maximum operations with the previous values, again in parallel.

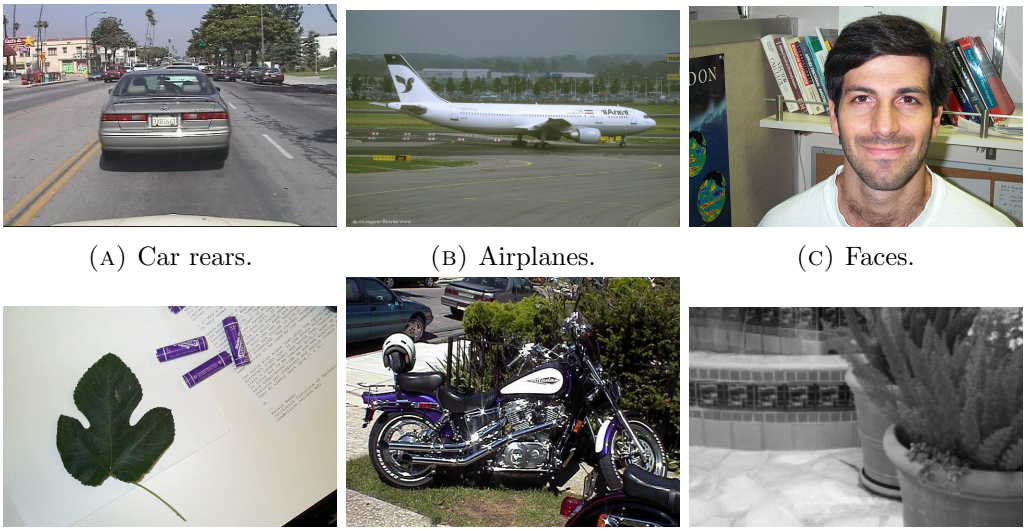
4.1.2 Results

That system all fits in the chosen Virtex-6 ML605 FPGA, including the temporary results and the pre-determined data that are stored in the device's BRAM. It was synthesized using Xilinx ISE tools. It has a latency of 600k clock cycles, with a throughput of one image every 526k clock cycles. The system may operate at 100MHz, with implies a 6ms latency and a 190 image per second throughput. The total resource utilization of the device is given in Table 4.1.
Finally the VHDL implementation was tested on binary classification tasks, using 5 classes of objects from Caltech101 and a background class. Accuracies for those tasks are given in Table 4.2. Results show that the accuracy on FPGAs is comparable to that of CPU implementations.

In this Section, we presented the work proposed by Orchard *et al* [99] and the architecture of their implementation. Next Section is dedicated to our contribution, which mainly consists of reducing the precision of the data throughout the process.

Category	Original model	CPU	FPGA
Airplanes	96.7	97.1	98.2
Cars	99.7	99.3	99.2
Faces	98.2	95.8	96.4
Leaves	97.0	94.6	93.7
Motorbikes	98.0	98.3	98.8

TABLE 4.2: Accuracies of Orchard's implementations on Caltech101 [99]. The "Original model" column shows the results obtained with the original HMAX code, while "CPU" shows the results obtained by Orchard *et al.*'s own CPU implementation, and "FPGA" show the results obtained with their FPGA implementation.



(A) Car rears. (B) Airplanes. (C) Faces.

(D) Leaves. (E) Motorbikes. (F) Background.

FIGURE 4.1: Samples of images of the used classes from Caltech101 dataset [142].

4.2 Proposed simplification

In order to save hardware resources, we propose several optimizations to the original HMAX model. Our approach mainly consists in simplifying the encoding of the data and reducing the required number of bits. In order to determine optimal encoding and algorithmic optimizations, we test each of our proposition on the widely used Catlech101 dataset. For fair comparison with other works, we use the same classes as in [99]: "airplanes", "faces", "car rear", "motorbikes" and "leaves".

Optimizations are tested individually, starting from those intervening at the beginning of the feed-forward and continuing in processing order, to finish with optimizations to apply to the later layer of the model. For optimizations having tunable parameters (e.g the bit width), those tests shall be used to determine a working point, which is done for all optimizations that require it in order to have a complete and usable optimization scheme. Optimizations are performed at the following levels: the input data, the coefficients of the Gabor filters in S1, the data produced by S1, the number of filters in S2, and finally

FIGURE 4.2: Precision degradation in input image for three types of objects: faces, cars and airplanes. Color maps are modified so that the 0 corresponds to black and the highest possible value corresponds to white, with gray level linearly interpolated in between. We can see that while the images are somewhat difficult to recognize with 1 bit pixels, they are easily recognizable with as few as 2 bits.

the computation of the distances in S2 during the pattern matching operations. We shall first present our work, namely the reduction of the precision of the input pixels. We shall then see how that optimization behaves with further optimizations got from the literature.

4.2.1 Input data

Our implementation of HMAX, along the lines of what is done in [168], processes grayscale images. The pixels of such images are typically coded on 8 bits unsigned integers, representing values ranging from 0 to 255, where 0 is "black" and 255 is "white". We propose here to use less than 8 bits to encode those pixels, simply by keeping the Most Significant Bits (MSB). This is equivalent to an Euclidean division by a power of two: unwiring the N Least Significant Bits (LSB) amounts to perform an Euclidean division by 2^N. The effect of such precision degradation is shown in Figure 4.2.

In order to find the optimal bit width presenting the best compromise between compression and performance, an experiment was conducted. It consisted of ten independent *runs*. In each run, the four classes are tested in binary independent classification *tasks*. Each task consists in splitting the dataset in halves: one half is used as the training set, and the other half is used as the testing set. All images are resized so that their height

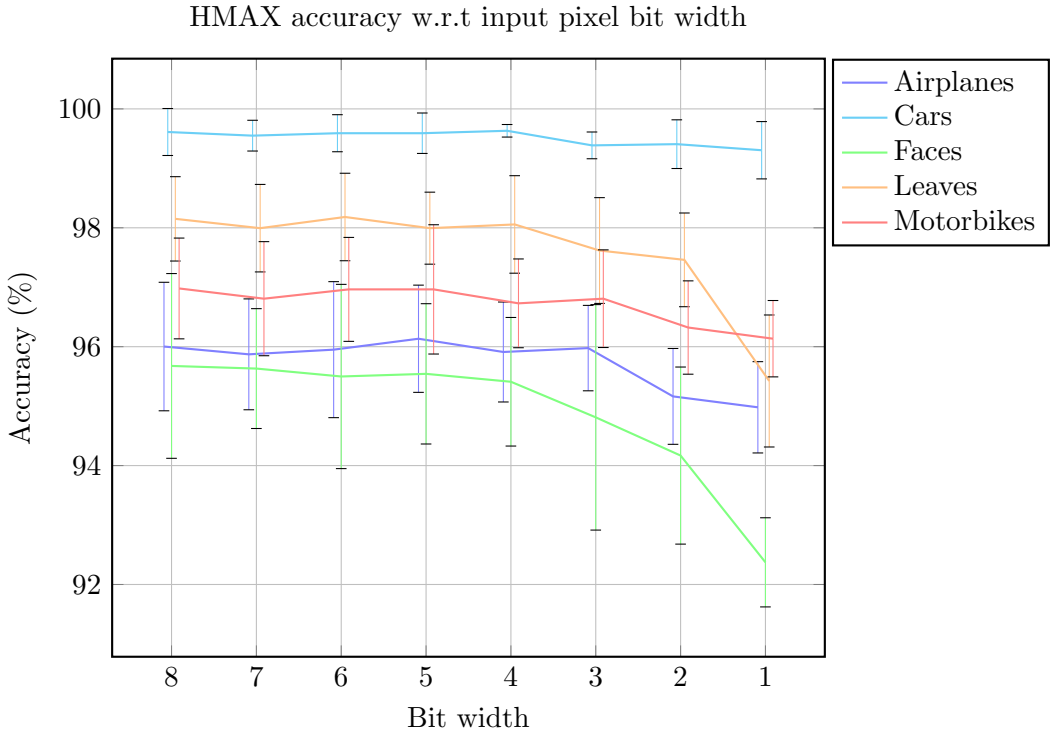


FIGURE 4.3: Recognition rates of HMAX on four categories of Caltech101 dataset w.r.t the input image pixel bit width. For each bit width, ten independent tests were carried out, in which half of the data was learnt and the other half was kept for testing. We see that the pixel precision has little to no influence on the accuracy.

is 164 pixel, and are then degraded w.r.t the tested bit width, i.e. all pixels are divided by 2^N where N is the number of removed LSB. The degraded data is then used to train first HMAX, and then the classifier – in this case, GentleBoost [169]. The images used as negative samples are taken from the Background Google class of Caltech101. All tests were performed in Matlab. Is should also be noted that we do not use RBFs in the S2 layer as described in [168] and in Section 2.1.2.2.The global accuracy for each class is then given by the mean of the recognition rates for that class across all runs, and the uncertainty in the measure is given by the standard deviations of those accuracies. Finally, the random seed used in the pseudo-random number generator was manually set to the same value for each run, thus ensuring that the conditions across all bit-widths are exactly the same and only the encoding changes.

The results of this experiment are shown in Figure 4.3. It is shown that for all four classes the bit width has only limited impact on performances: all accuracies lie beyond 0.9, except when the input image pixels are coded on a single bit where the Airplanes class gets more difficult to be correctly classified. For that reason, we chose to set the input pixel's bit width to 2 bits, and all further simplifications shall be made taking that into account. The next step is to reduce the precision of the filter's coefficient, in a way that is somewhat similar to what is proposed in [167].

4.2.2 S1 filters coefficients

The second simplification that we propose is somewhat similar to that presented in Section 4.2.1, except this time we operate on the coefficients of the Gabor filters used in S1. Mathematically, those coefficients are real numbers in the range $[-1, 1]$, thus the most naive implementation for them is to use double precision floating point representation as used by default in Matlab, and that encoding scheme shall be used as the baseline of our experiments. Our simplifications consist in using signed integers instead of floats using n-bits precision, by transforming the coefficients so that their values lie within $\left\{-2^{n-1}, \ldots, 2^{n-1} - 1\right\}$, which is done by multiplying them by 2^{n-1} and rounding them to the nearest integer. Several values for n where tested, along the lines of the methodology described in Section 4.2.1: 16, and from 8 downto 1. However, using the standard signed coding scheme the 1 bit encoding would lead to coefficients equal either to -1 or 0, which does not seem relevant in our case. Thus, we proposed to use a particular coding here, where the "0" binary value actually encodes -1 and "1" still encodes 1. The rational is that that encoding is close to the Haar-like features used in Viola-Jones [30] as explained in Section 3.1.1.1, and this technique is also suggested in [170]. As explained in Section 4.2.1, the input pixels precision is 2 bits.
Recent works [171] also propose much more sophisticated encoding scheme. While their respective efficiencies have been proven, they seem more adapted to a situation where the weights are learnt during the learning process, and thus unknown before learning. In our case, all weights of the convolution are predetermined, thus we have a total control over the experiment and we prefered to use optimizations as simple as possible.

Results for that experiment are given in Figure 4.4. We see that the impact of the encoding of the Gabor filter coefficients has even less impact than the input image pixels precision, even in the case of 1 bit precision. This result is consistent with the fact that Haar-like features are used with success in other frameworks. Thus, we shall use that 1 bit precision encoding scheme for Gabor filters in combination with the 2 bit encoding for input pixels in further simplifications.

In this Section, we validated that we could use only one bit to encode the Gabor filter's coefficients, using "0" to encode "-1" and "1" to encode "1", in conjunction with input pixels coded on two bits only. In order to continue our simplification process, next Section proposes optimizations concerning the output of S1.

4.2.3 S1 output encoding

It has been proposed in [167] to use Lloyd's algorithm [172, 173], that provides a way to find an optimal encoding w.r.t a desired number K of possible quantization and a

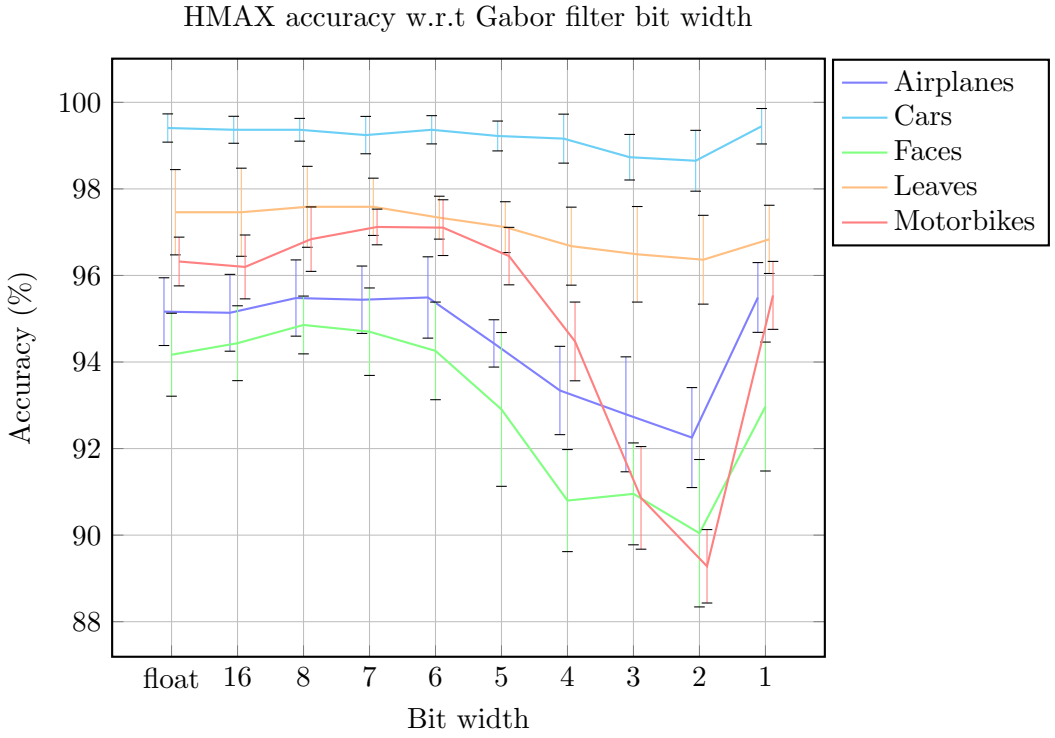


FIGURE 4.4: Recognition rates on four categories of Caltech101 dataset w.r.t the coefficients of the Gabor filter coding scheme in S1 layer. Those tests were run with input pixels having 2 bits widths. The protocol is the same as developped for testing the input pixels, as done in Figure 4.3.

.

subset S of the data to encode. The encoding strategy consists in defining two sets: a *codebook* $C = \{c_1, c_2, \ldots, c_Q\}$ and a *partition* $Q = \{q_0, q_1, q_2, \ldots, q_{K-1}, q_K\}$. With those elements, mapping a code $l(x)$ to any arbitrary value $x \in \mathbf{R}$ is done as follows:

$$\forall x \in \mathbf{R} \quad l(x) = \begin{cases} c_1 & x \leq q_1, \\ c_2 & q_1 < x \leq q_2, \\ \ldots & \\ c_{q-1} & q_{K-2} < x \leq q_{K-1}, \\ c_q & q_{K-1} < x. \end{cases} . \tag{4.12}$$

One can see here that p_0 and p_K are not used to encode data; however those values are required to be computed when determining the partition, as we shall now see.
Finding the partition consists in minimizing the Mean Square Error $E(C, P)$ between the real values in the subset and the values after quantization [167, 172]:

$$E(C, P) = \sum_{i=1}^{K} \int_{q_{i-1}}^{q_i} |c_i - x|^2 \, p(x) \, dx \tag{4.13}$$

Where p is the *probability distribution* of x. One can show [167] that

$$\forall i \in \{1, \ldots, K\} \quad c_i = \frac{\int_{q_{i-1}}^{q_i} xp\left(x\right) dx}{\int_{q_{i-1}}^{q_i} p\left(x\right) dx} \tag{4.14}$$

$$\forall i \in \{1, \ldots, K-1\} \quad q_i = \frac{c_{i-1} + c_i}{2} \tag{4.15}$$

$$a_0 = \min S \tag{4.16}$$

$$a_K = \max S \tag{4.17}$$

We see that Equations 4.14 and 4.15 depend on each other, and there is no closed-form solution for them. The optimal values are thus determined with an iterative process: starting from arbitrary values for $Q = \{q_1, q_2, \ldots, q_K\}$, e.g separating the range of values to encode in segments of same size:

$$\forall k \in \{1, \ldots, \} \, q_i = q_0 + k\frac{q_K - q_0}{K}, \tag{4.18}$$

we compute $C = \{c_1, \ldots, c_k\}$ with Equation 4.14. Once this is done, we use those values to compute a new ensemble Q with Equation 4.15, and so on until convergence.

Since the dynamics of the values vary greatly from scales to scales in C1, we computed a set C_i and Q_i per C1 scale in i. However, contrary to what is proposed in [167], we did not separate the orientations. We thus produced 8 sets S_i of data to encode ($i \in \{1, \ldots, 8\}$). using the same 500 images selected at random among all of the five classes we use to test our simplifications. As suggested in [167], we used four quantization levels for all S_i. Each partition Q_i and code book C_i where computed using Matlab's Communication System Toolbox's `lloyd` function. The results are given in Table 4.3. While this simplification uses the values computed in C1, it is obvious that it could easily be performed at the end of the S1 stage, simply by using a strictly growing encoding function f. This is easily performed by associated each value from C_i to a positive integer as follows:

$$\forall i \in \{1, \ldots, 8\}, j \in \{1, \ldots, 4\} \, f\left(c_i j\right) = j \tag{4.19}$$

and encoding $f\left(c_{ij}\right)$ simply as unsigned integers on 2 bits. By doing so, performing the max-pooling operations in C1 after that encoding is equivalent to performing it before.

We must now make sure that this simplification, in addition to the other two presented earlier, does not have a significant negative impact on accuracy. Thus, we perform an experiment along the lines of what is described in Section 4.2.1, with the exception that this time we add the simplification proposed here. Results are compiled with further optimizations in Table 4.4

i	1	2	3	4
C_1	14	27	37	50
Q_1	21	32	43	-
C_2	42	82	118	154
Q_2	62	100	136	-
C_3	37	65	94	141
Q_3	51	79	117	-
C_4	81	148	209	284
Q_4	114	178	246	-
C_5	122	208	278	380
Q_5	165	243	329	-
C_6	175	309	427	559
Q_6	242	368	494	-
C_7	296	521	707	905
Q_7	408	614	806	-
C_8	499	868	1182	1492
Q_8	633	1025	1337	-

TABLE 4.3: Code books and partitions by scales for features computed in C1. Values were computed with the simplification proposed in Sections 4.2.1 and 4.2.2 for S1, using Matlab's `lloyds` function.

4.2.4 Filter reduction in S2

As it has been stated many times in the literature [91–96, 98, 99], the most demanding stage of HMAX is S2. Assuming there are the same amount of pre-learnt patches of each size, then the algorithmic complexity depends linearly on the amount of filters N_{S2} and their average number of elements $\overline{K}$. It has been suggested in [46] to simply reduce the number of per-learnt patches in S2 by sorting them by *relevance* according to a criterion, and to keep only the N most relevant patches. The criterion used by the authors is simply the variance ν of the components inside a patch $p = (p_1, \ldots, p_M)$:

$$\nu(p) = \sqrt{\sum_{i=1}^{M} |p_i - \overline{p}|^2}, \tag{4.20}$$

$$\overline{p} = \frac{1}{M} \sum_{i=1}^{M} p_i. \tag{4.21}$$

In their paper, Yu *et al.* proposed to keep the 200 most relevant patches, which when compared to the 1000 patches recommended in [168] would allow to divide the complexity at this stage by 5. In [168], it is suggested to use patches of 4 different sizes: $4 \times 4 \times 4$, $8 \times 8 \times 4$, $12 \times 12 \times 4$ and $16 \times 16 \times 4$.

In order to ensure that all sizes are equally represented, we propose to first crop at

random 250 patches of each of those sizes in order to get the suggested 1000 patches by Serre *et al.* [168], and we select 50 patches of each size according to the variance criterion so that we have a total of 200 patches, as proposed in [46]. The rational is that we must keep in mind that we aim to implement that process on a hardware device, thus we need to know in advance the amount of patches of each size and to keep them to pre-determined values.

Let's now experiment that simplification on our dataset. We followed the methodology established in Section 4.2.1, and we used the simplification proposed here along with all the other simplifications that were presented until now. Results are compiled with those of Section 4.2.3 and Section 4.2.5 in Table 4.4.

4.2.5 Manhattan distance in S2

In S2, pattern-matching is supposed to be performed with a Gaussian function, the centers of which are the pre-learnt patches in S2, so that each S2 unit returns a value close to 1 when the patterns are closed in terms of Euclidean distance and 0 when they are far from each others. Computing an Euclidean distance implies the computation of square and square-roots function, which may use lots of hardware resources. The evaluation of the exponential function also raises similar issues, along with those already exposed in Section 4.2. Since we already removed the Gaussian function to simplify the training of S2, we propose to compare the performances obtained when replacing the Euclidean distance with the Manhattan distance:

$$M\left(v_1, v_2\right) = \sum_{i=1}^{N_v} \left|v_{1i} - v_{2i}\right|. \tag{4.22}$$

Doing so allows to remove all multiplications, which simplifies further the implementation on FPGA. Results are compiled with those of Section 4.2.3 and Section 4.2.4 in Table 4.4.

In this Section, we proposed a series of optimizations, both of our own and from the literature. In the next Section, we show how that particular encoding may be put into practice on a dedicated hardware configuration.

	Input and filter coefficients	Lloyd's encoding	Filter reduction in S2	Manhattan distance
Airplanes	95.49 ± 0.81	94.43 ± 0.88	92.07 ± 0.69	91.83 ± 0.63
Cars	99.45 ± 0.41	99.35 ± 0.40	98.45 ± 0.54	98.16 ± 0.60
Faces	92.97 ± 1.49	90.11 ± 1.05	82.71 ± 1.32	83.35 ± 1.40
Leaves	96.83 ± 0.79	97.21 ± 0.89	94.61 ± 1.12	93.20 ± 1.42
Motorbikes	95.54 ± 0.79	94.79 ± 0.62	88.83 ± 1.10	89.08 ± 1.31

TABLE 4.4: Accuracies of HMAX with several optimizations on five classes of the Caltech101 dataset [142]. That Table compiles the results of the experiment conducted in Sections 4.2.3, 4.2.4 and 4.2.5. The column on the left shows the result gotten from Section 4.2.2. Starting from the second column, each column show the accuracies obtained on the 5 classes in binary task classification, as described before, taking into account the corresponding simplification as well as those referred by the columns left to it.

4.3 FGPA implementation

4.3.1 Overview

We propose now our own implementation of the HMAX model, using both our contributions and the simplifications proposed in the literature proposed in Section 4.2. We did not use the architectural optimization proposed in [99] on purpose, to see how a "naive" implementation of the optimized HMAX model compares with that of Orchard *et al.* This implementation of the HMAX model with our optimizations intends to process fixed-size grayscale images. We aim to process 164×164 grayscale images. The rational behind those dimensions is that we want to actually process the 128×128 ROI located at the center of the image – however, the largest convolution kernel in S1 is 37×37, therefore in order to have 128×128 S1 feature maps we need input images padded with 18 pixels wide stripes. That padding is assumed to be performed before the data is sent to the HMAX module.

The data is processed serially, i.e pixels arrive one after the other, row by row. The pixels' precision is assumed to be already reduced to two bits per pixel, as suggested in Section 4.2. The module's input pins consists in a serial bus of two pins called `din` in which the pixels should be written, a reset pin `rst` allowing to initialize the module, an enable pin `en_din` allowing to activate the computation and finally three clocks: a "pixel clock" `pix_clk` for input data synchronization, a "process clock" `proc_clk` synchronizing the data produced by the module's processes, and a "sub-process clock" `subproc_clk` as some processes need a high-frequency clock. Suggestion concerning the frequencies of those clocks are given in Section 4.4.
The output pins consist in: an 8 pins serial bus for the descriptor itself called `dout` and a pin indicating when data is available named `en_dout`.

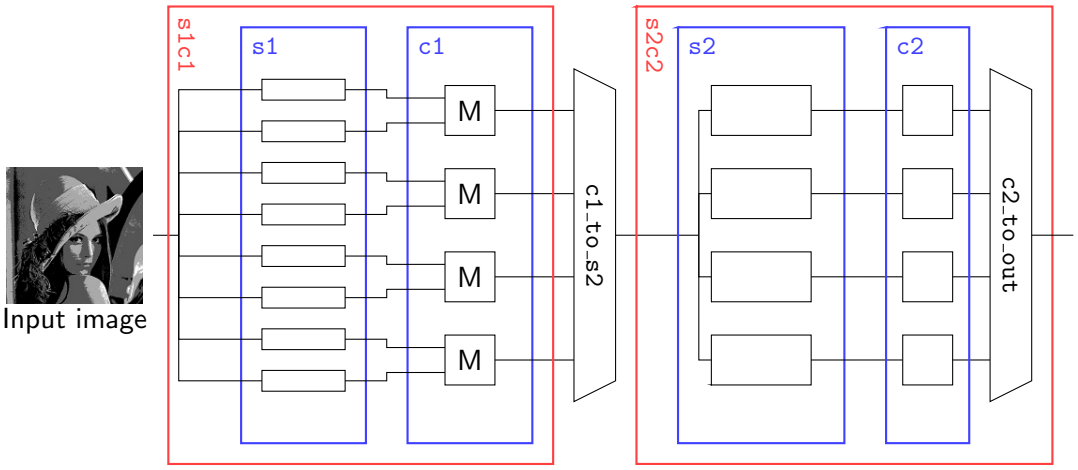


FIGURE 4.5: HMAX VHDL module. The main components are shown in colors, and the black lines represent the data flow. We see here that the data from the degraded 164×164 input image is first processed by S1 filters at all scales in parallel – only 8 out of the 16 filters in the bank are shown for readability. Orientations are processed serially and the outputs are multiplexed. The data is then processed by the `c1` module, which produces half the feature maps produced in S1, before being serialized by `c1_to_s2`. The serialized data is sent to `s2c2`, which perform pattern matching between input data and pre-learnt patches with its `s2` components, several in parallel, with a multiplexing. The maximum responses of each S2 unit are then computed by `c2`. The data is then serialized by `c2_to_out`.

The HMAX module – illustrated in Figure 4.5 – itself mainly consists in two sub-modules, `s1c1` and `s2c2`. As suggested in their names, the first one performs the computations required in the S1 and C1 layers, while the second one takes care of the computation for the S2 and C2 layers of the model. The rational behind that separation is that it is suggested in [31] that in some cases one may use only the S1 and C1 layers, as we did in Chapter 3. The following two Sections describe those modules in detail.

4.3.2 `s1c1`

That module consists uses two components of its own, called `s1` and `c1`, which performs the operations required by the layers of the same names of the model. It process the input pixels with a multiplexing across orientations, meaning that all processes concerning the first orientation of the Gabor filters in S1 are performed in the same clock cycle, then all processes concerning the second orientation are performed *on the same input data*, and so on until all four orientations are processed.

The input pins of that module are directly connected to those of the top module. Its input pins consist in a `dout` bus of 4 pins where the C1 output data are written, a `en_dout` pin indicating when new data is available and a `dout_ori` serial bus that precises which orientation the output data corresponds to. The `s1` and `c1` modules shall now be presented.

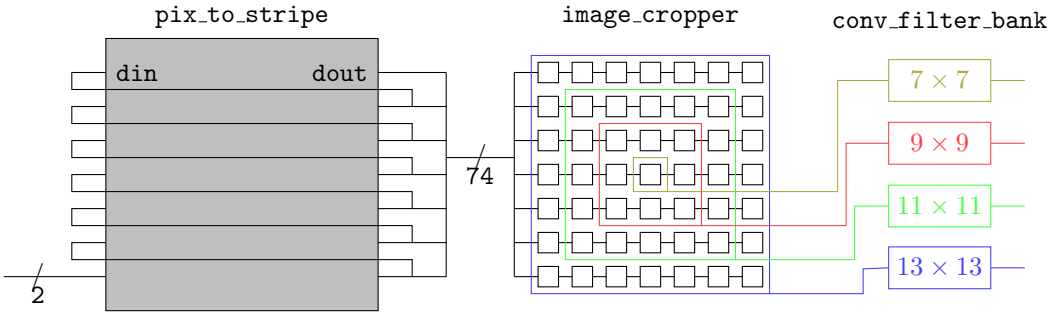


FIGURE 4.6: Dataflow in `s1`. This Figure shows the major components of the `s1` module. First of all the pixels arrive in the `pix_to_stripe`, which returns columns of 37 pixels. Those columns are then stored in shift registers, which store a 37×37 patch – only 7 lines are represented here for readability. Then for each of the 16 scales in S1, there exists an instance of the `image_cropper` module that keeps only the data needed by its following `conv` module. The convolution kernels' coefficients are gotten from the `coeffs_manager` module, which get them from the FPGA's ROM and retrieve those corresponding to the needed orientation, for all scales.
Here only 4 of the 16 convolution engines are shown. The computed data is written in `dout`, in parallel. Note that not all components of `s1` are repesented here: `pixmat`, `pixel_manager`, `coeffs_manager` and `conv_crop` are not displayed to enhance readability and focus and the dataflow.

4.3.2.1 `s1`

That module consists in three sub-modules: `pixel_manager` which gets the pixels from the input pins and reorder them so that they may be used in convolutions, the `coeffs_manager` module which handles the coefficients used in the convolution kernels, and the convolution filter bank module `conv_filter_bank` which take care of the actual linear filtering operations. Shift registers are also used to synchronize the data produced by the different components when needed. The main modules are described below, and the dataflow in the module is sum up Figure 4.6.

`pixel_manager` As mentioned in Section 4.3.1, the data arrives in our module serially, pixel by pixel. It is impractical to perform 2D convolutions in those conditions, as we need the data corresponding to a sub-image of the original image. The convolution cannot be processed fully until all that data arrives, and the data not needed at a particular moment needs to be stored. This is taken care of by this component: it stores the temporary data and outputs it when ready, as a 37×37 pixel matrix as needed by the following `conv_filter_bank`, as explained below. That process is performed by two different sub-modules: `pix_to_stripe`, which reorder the pixels so that they may be processed column per column, and the `pixmat` that stores the data in a matrix of registers and provide them to the convolution filter bank module.

pix_to_stripe That modules consists in a BRAM, the output pins of which are rewired to its input pins in the way shown in Figure 4.6. It gets as inputs, apart from the usual `clk` , `en_din` and `rst` pins, the 2 bit pixels got from the top-module. Its output pins consist in a $37 \times 2 = 74$ pins bus providing a column of the 37 pixels, as well as a `en_dout` output port indicating when data is ready to be processed.

pixmat That module gets as inputs the outputs of the aforementioned `pix_to_stripe` module. It simply consists in a matrix of 37×37 pixels. At each pixel clock cycle, all registered data is shifted to the "right", and the registers on the left store the data gotten from `pix_to_stripe`. The `pixmat` module's output pins are directly wired to its outputs, and an output pin called `en_dout` indicates when the data is ready. When that happens, the data stored in the matrix of registers may be used by the convolution engines.
In order to handle new lines, that module has an inner counter incremented every time new data arrives. When that counter reaches 164, i.e when a full stripe of the image went through the module, the `en_dout` signal is unset and the counter is reset to 0. The `en_dout` signal is set again when the counter reaches 37 again, meaning that the matrix is filled.

coeffs_manager That module's purpose is to provide the required convolution kernels' coefficient, w.r.t the required Gabor filters orientation. It gets as inputs the regular `rst`, `clk` and `en` signals, but also a bus of two pins called `k_idx` indicating the desired orientation The output pins consists of the customary `en_dout` output port indicating that the data is ready, and a large bus called `cout` that outputs *all* coefficients of *all* scales for the requested orientation. This is also close to the box filter approximation proposed in [167]. As explained in Section 4.2, we use a particular one bit encoding. Since our convolution kernels' sizes go from 7×7 to 37×37 by steps of 2×2, the total amount of input pins in the `cout` bus is given by

$$\sum_{k=0}^{15} (2k+7)^2 = 9104 \tag{4.23}$$

All the coefficients are stored in BRAM. The module fetches the needed ones depending on the value written in `k_idx`, and route them to the `cout` module. Figure 4.7 illustrates that module.

conv_filter_bank This module gets its inputs directly from the `pixmat` and `coeffs_manager` modules just described. The pixel matrix is written in the `din` input bus, and the coefficients used for the convolution are written in the `coeffs` bus. It also has the usual

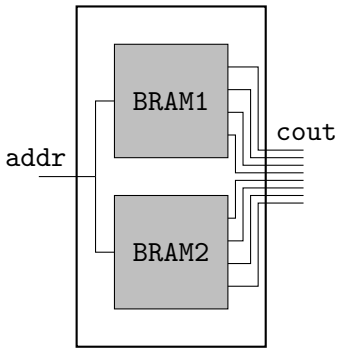


FIGURE 4.7: `coeffs_manager` module. In order to simplify the process, all coefficients needed at a particular time are read all at once from several BRAM, of which only two are represented here for readability. The coefficients are then concatenated in a single vector directly connected to the `cout` output port.

`en_din` and `rst` pins, which serve their usual purposes. It also gets the orientation identifier thanks to an `id_in` input bus – that identifier is not directly used for computation, but is passed with the output data for use in latter modules. Finally, that modules needs two clocks: the *pixel clock*, on which the input data is synchronized and acquired through the `clk` pin, and the *process clock* (acquired through `clk_proc`) needed for multiplexing the filters per orientations, as suggested in Section 4.3.1.
Output pins consist in a `dout` bus in which the result of the convolutions at all scales are written, an `id_out` bus simply indicating the orientation identifier got from the `id_in` input bus and the usual `en_dout` pin. In order to perform its operations, that module has one distinct instance of the `conv_crop` component per scale (i.e, 16 instances in total). Each instance has parameters of its own depending on its scale.

`conv_crop` That module's input and output ports are similar to those of its parent module `conv_filter_bank`. It gets the pixel and process clocks respectively from its `clk` and `clk_sum` input ports, and it may be reset using the `rst` input port. Image data arrive through `din`, and the convolution coefficients got from `coeffs_manager` are acquired through the `coeffs` input port. Data identifier is given by `id_in` input port, and `en_din` indicates when input data is valid and should be processed. Output ports encompass `dout`, which provide the results of the convolution, and `id_out` which gives back the signal got from `id_in`. Finally, `en_dout` indicates when valid output data is available. `dout` signals from all instances of `conv_crop` are then gathered in `conf_filter_bank`'s `dout` bus. This module gets its name from its two main purposes: select the data required for the convolution, and perform the actual convolution.
The first stage is done asynchronously by a component called `image_cropper`. As explained earlier, `conv_crop` get the data in the form of a 37×37 pixel matrix – however, all that data is only useful for the 16th scale convolution kernel, which is also of size 37×37. A $N \times N$ convolution kernel need only the pixels in the $N \times N$ sub matrix

located in the middle of the 37×37 matrix, as shown in Figure 4.6. The selected data is then processed by the `conv` component, which is detailed in the next section.

4.3.2.2 `conv`

That module carries out the actual convolution filter operations. It gets as inputs two clocks: `clk` which gets the *process clock* and `clk_sum` which is used to synchronize sums in the convolution *sub-process clock*. It also has the usual `rst` pin for initialization, a bus called `din` through which the pixel matrix arrives, a bus called `coeffs` which gets the convolution kernel's coefficients, an `id_in` bus allowing to identify the orientation that is being computed, and an `en_din` pin warning that the input data is valid and that operations may be performed. Its outputs are a `dout` bus that provides the convolution results, another one called `id_out` that indicates which orientation that data corresponds to and a `en_dout` bus announcing valid output data.

In order to simplify the architecture and to limit the required frequency of the sub-process clock, the convolution is first performed row by row in parallel. The results of each rows are then added to get the final result. That row-wise convolution is performed by a bank of `convrow` module having one filter per row. The sum of the rows are performed by the `sum_acc` module, and the result is coded as suggested in Section 4.2 thanks to the `s1degrader` module; both modules shall now be presented.

`convrow` That module has almost the same inputs as `conv`, the only exception being that it only gets the input pixels and coefficients corresponding to the row it is expected to process. Its output pins are similar to those of `conv`. As explained in Section 4.2, our filters coefficients are either $+1$'s and -1's, respectively coded as "1" and "0". Thus, each 1 bit coefficient does actually not code a value, but rather an *instruction*: if the coefficient is 1, the corresponding pixel value is added to the convolution's accumulated value, and it is subtracted if the coefficient is 0. That trick allows to perform the convolution without any products. In practice, a subtraction is performed by getting the opposite value of the input pixel by evaluating its two's complement and performing and addition. Sums involved at that stage are carried out by the `sum_acc` module, which shall now be described.

`sum_acc` That module sums serially the values arriving in parallel. The data arrives through its `din` parallel bus, and must be synchronized with the *process clock* arriving through the `clk` pin. That module uses a unique register to store its temporary data. At each *process clock* cycle, the MSB of the `din` bus, which correspond to the first value of

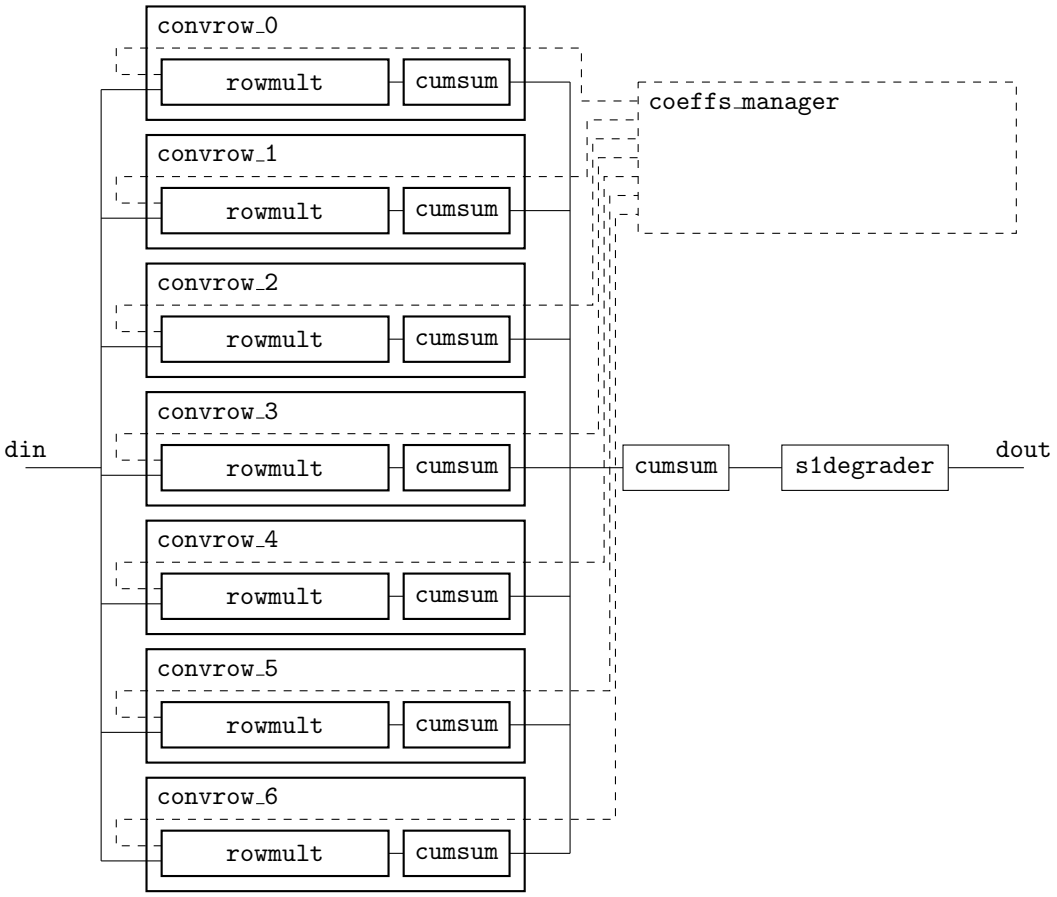


FIGURE 4.8: 7×7 convolution module. That module has one `convrow` module per row in the convolution kernel, each taking care of a line. In each of those modules, the "multiplications" are performed in parallel in `rowmult` between the data coming from `din` and `coeffs` input buses – as mentioned in Section 4.2, those multiplications consist in fact in simple changes of signs, depending on the 1 bit coefficients provided by the external module `coeffs_manager`. The results are the accumulated thanks to `convrow`'s `cumsum` component. Finally, the output of all `conrow` modules are accumulated thanks to another `cumsum` component. The result is afterward degraded thanks to the `s1degrader` module, the output of which is written in `dout`.

the sum, is written in the register. At each following *sub-process clock* cycle, an index is incremented, indicating which value should be added to the accumulated total. Timing requirements concerning the involved clocks are discussed later in Section 4.4.2. The result is written on the output pins synchronously with the *process clock*.

Once the data has been accumulated row by row, and the results coming out of all rows have been accumulated again, the result may be encoded on significantly shorter words as we explained in Section 4.2.3. That encoding is taken care of by the `s1degrader` module, which shall be described now.

`s1degrader` This modules takes care of the precision degradation of the convolution's output. It is synchronized on the *process clock*, and as such has a `clk` input pin, and

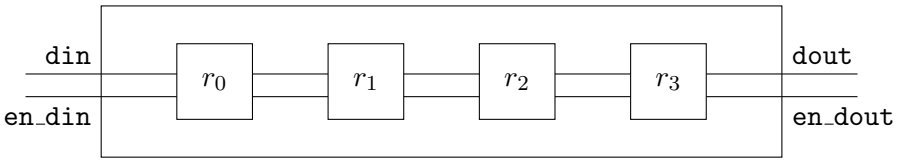


FIGURE 4.9: `shift_registers` module with 4 registers. At each clock cycle, data is read from `din` and `en_din` and written into the next register, the last of which writes its data into `dout` and `en_dout` output ports.

a `rst` pin for initialization. Input data feed this module through its `din` input port, and the `en_din` single-bit input port indicates the presence of valid data in `din`. The code is written into `dout` and `en_dout` warns the other modules that valid output data is available.

The computation is very simple. It simply consists in comparing the input data with the partition determined for that scale with the Lloyd algorithm presented in Section 4.2.3. The results written in `dout` simply depends on the position of the input value w.r.t the partition boundaries on the natural integer line.

4.3.2.3 `shift_registers`

That module allows to delay data. This is mostly useful to address synchronization problem, and thus it needs a clock `clk`. A `rst` input port allows to initialize it, and data is acquired through the `din` port while an `en_din` input port allows to indicate valid input data. Delayed data may be read from the `dout` output port, and a flag called `en_dout` is set when valid output data is available and unset otherwise.

The way that module works is straightforward. It simply consist in N registers r_i, each one of them being connected to two neighboors except for r_1 and r_N. At each clock cycle, both the data from `din` and `en_din` are written in r_1, and each other register r_i gets the data from its neighboor r_{i-1} as shown in Figure 4.9. The last register simply writes its data in the `dout` and `en_dout` output ports.

4.3.2.4 `c1`

Once the convolutions are done and the data encoded on a shorter word, max-pooling operations must be performed. Following the lines of the theoretical model, this is done by the `c1` module, which gets its inputs directly from `s1` output pins. It is synchronized on the *process clock*, and therefore it has the mandatory `clk` and `rst` pins. It also has input buses called `din`, `din_ori` and `en_din` which are respectively connected to `s1`'s `dout`, `ori` and `en_dout`. Its outputs pins are made up of buses named `dout`, `dout_ori`

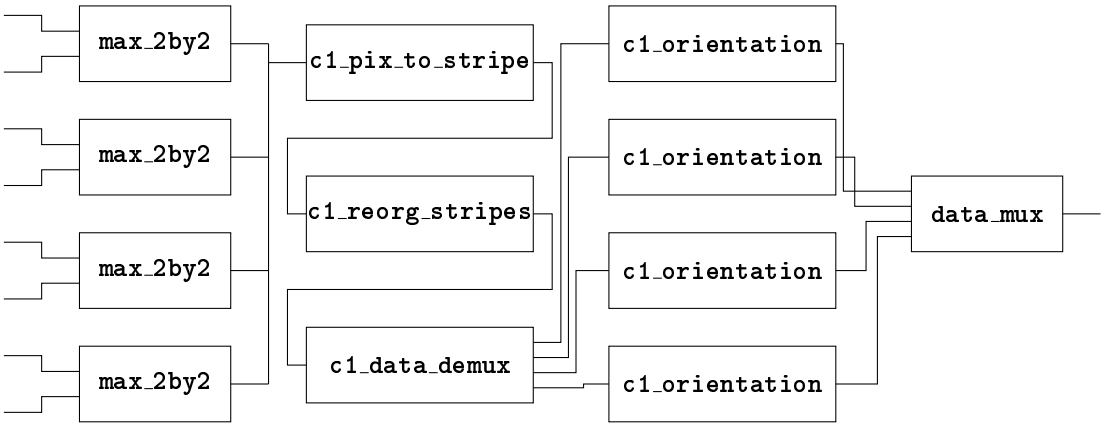


FIGURE 4.10: `c1` module. For more readability, only 4 of the 8 filters are represented here. Maximums are first computed accross scales with the `max_2by2` components. The data is then organized into stripes in the same fashion as done in the `pix_to_stripe` component used in `s1` module. That stripe is organized by lines, and then scales, and needs to be organized by scales, and then lines to be processed by the latter module – this reorganization is taken care of by `reorg_stripes`. Orientations being multiplexed, we needed to separate them so each may be processed individually, which is done by the `data_demux` module. Each orientation is then processed by one of the `c1_orientation` module. Finally, data comming out of `c1_orientation` is multiplexed by `data_mux` before being written in output ports.

and `en_dout`, which respectively provide the result of the max-pooling operations, the associated orientation identifier and the flag indicating valid data.

The process is carried out by the following components: `c1_max_2by2` which computes the pixel-wise maximum across two S1 feature maps of consecutive scales and same orientation, `c1_pix_to_stripe` which reorganize the values in a way similar to that of the aforementioned `pixel_manager` module, `c1_reorg_stripes` which routes the data to the following components in an appropriate manner, `c1_orientation_demux` which routes the data to the corresponding max-pooling engine depending on the orientation it corresponds to, and finally `max_filter` which *is* the actual max-pooling engine and performs for a particular orientation, hence the name. That flow is shown in Figure 4.10.

`c1_max_2by2` Apart from the `clk`, `rst` and `en_din` input pins, that module has an input bus called `din` that gets the data produce by all convolution engines and perform the max-pooling operations across consecutive scales. Since the immediate effect of that process is to divide the number of scales by two, that module's output bus `dout` has half the width of `din`. A signal going through the `en_dout` output pin indicates that valid data is available *via* `dout`.

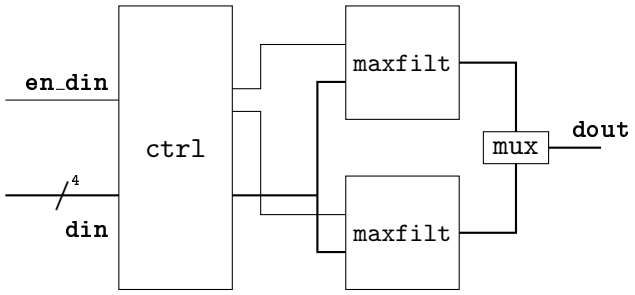
`c1_pix_to_stripe` That module is very similar to the `pix_to_stripe` module used in `s1` (see Section 4.3.2.1), except that it operates on data of all of the 8 scales produced by

`c1_max_2by2` and produces stripes of 22 pixels in heights, as the maximum window used for the max-pooling operations in C1 is 22×22 as stated in [31]. Its input and output ports are the same as those of `pix_to_stripe`, with additional `din_ori` and `dout_ori` allowing to keep track of the orientation corresponding to the data.

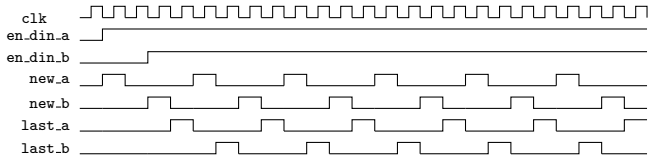
`c1_reorg_stripes` The data produced by `c1_pix_to_stripe` is ordered first by the position of the pixels in its stripe, and then per scale – i.e first pixels of all scales are next to each others, followed by the second pixels of all scales, and so on. This is impractical for the processed needed in the later module, where we need the data to be grouped by scales. That module achieves it simply by rerouting the signals asynchronously.

`c1_orientations_demux` During C1, each orientation is performed independently from the others. However, at this point they arrive multiplexed from the same bus: first pixels from the first orientation, then the pixels at the same locations from the second orientation, followed by the third and the fourth – we then go back to the first orientation, then the second one and so on. That modules gets those pixels through its `din` bus, and route the signal to the relevant pins of its `dout` bus depending on its orientation, which is given by the `din_ori` input bus, which is wired to `c1_pix_to_stripe`'s `dout_ori` bus. Each set of pins corresponding to a particular orientation then routes the signal to the correct instance of the `c1_orientation` module. In order to perform that demultiplexing operation, that module also has the compulsory `clk`, `rst` and `en_din` pins.

`c1_orientation` The actual max-pooling operation is performed by the `c1unit` components contained in that module. Each `c1_orientation` instance has a bank of 8 `c1unit` instances, each having its own configuration so as to perform the max-pooling according to the parameters indicated in [31]. The role of the `c1_orientation` module is to serve as an interface between the max-pooling unit bank and the rest of the hardware model. As inputs, is has the usual `clk`, `rst` and `en_din` input pins as well as a `din` input bus. That bus gets the data of the corresponding orientation generated in the `s1` module. Data of all scales arrive in parallel, as a result of the previous modules. Data of each of the 8 scales is routed to a particular `c1unit` component, which shall be described soon. Output data is then written in the `dout` bus. An `en_dout` output is set to "1" when data is ready, and pins of an output bus called `dout_en_scales` are set depending on the scales at which the data is available, while the other pins are unset – e.g, is the output data correspond to the 1st and 4th scales of the C1 layer of the model, `dout_en_scales` shall get the value "00001001".



(A) Architecture.



(B) Control.

FIGURE 4.11: `c1unit`. Figure 4.11a shows the principle components of the module architecture, and Figure 4.11b shows the control signals enabling and disabling the data.
Figure 4.11a shows the two `c1unit` components and the control module `c1unit_ctrl` – named `ctrl` here for readability. Data coming out of those components are multiplexed in the same output port `dout`. The four bits data signal is shown with the thick line, and the control signals ares shown in light line. We see that dedicated control signals are sent to each `maxfilt` components, but also that both get the same data.
The control signals presented in Figure 4.11b show how the control allow to shift the data between the two units, in order to produce the overlap between two C1 units. We assume here that we emulate C1 units with 4×4 receptive fields and 2×2 overlap.

`c1unit` This is the core-module of the max-pooling operations – the purpose of all other modules in `c1` is mostly to organize and route data, and manage the whole process. Its inputs consist in the compulsory `clk`, `rst` and `en_din` pins and the `din` bus. Data are written to the usual `dout` and `en_dout` output ports. The max-pooling operations are performed by two instances of a component named `maxfilt`. The use of those two instances, latter refered to as `maxfilt_a` and `maxfilt_b`, is made mandatory by the fact that there is 50% overlapping between the receptive fields of two C1 unit in the original model. The data is always sent to both components, however setting and unsetting their respective `en_din` pins at different times emulates the behaviour of the set of C1 units operating at the corresponding orientation and scale: at the beginning of a line, only one of the two modules is enabled, and the other one gets enabled only after an amount of pixels equal to half the size of the pooling window (e.g the *stride*) as arrived. That behaviour is illustrated in Figure 4.11, and is made possible thanks to the `c1unit_ctrl` module. In the next two paragraphs, we first describe how `maxfilt` works, and then how it is controlled by `c1unit_ctrl`.

maxfilt This is where the maximum pooling operation actually takes place. That module operates synchronously with the *process clock*, and thus has the usual `clk`, `rst` and `en_din` input ports – data is got in parallel via the `din` input port. The input data corresponds to a column of values generated by `s1`, with the organization performed by the above modules. There are also two additionnal control pins called `din_new` and `din_last`, allowing to indicate the module that the input data is either the first ones of the receptive field, the last ones, or intermediate data. The value determined by the filter is written in the `dout` port, and valid data is indicated with the `en_dout` output port.

The module operates as follows. the module is enabled only when the `en_din` port is set to "1". It has an inner register R that shall store intermediate data. When `din_new` is set, the maximum of all input data is computed and the result is stored in R. When both `din_new` and `din_last` are unset and `en_din` is set, the maximum between all input values and the value stored in R is computed and stored back in R. Finally, when `din_last` is set the maximum value between inputs and R is computed again but this time it is also written in `dout` and `en_dout` is set to "1". Figure 4.11b shows how those signals should act to make that module work properly.

c1unit_ctrl That module's purpose is to enable and disable the two `maxfilt` components of `c1unit` when appropriate. It does so thanks to a process synchronized on the *process clock*, and thus has the customary `clk`, `rst` and `en_din` input ports. It gets the data that is to be processed in its parent `c1unit` module through its `din` input bus, and re-write to the `dout` output bus along with flags wired to the two `c1unit` components of its parent module, via four output ports: `en_new_a` and `en_last_a` which are connected to `maxfilt_a`, and `en_new_b` and `en_last_b` which are connected to `maxfilt_b`. `maxfilt_a` and `maxfilt_b` are the modules mentioned in the the description of `c1unit`, presented earlier.

4.3.2.5 `c1_to_s2`

That module's goal is to propose an interface between the output port of `c1` and the input ports of `s2`. It also allows to get the data directly from `c1` and use it as a descriptor for the classification chain. It reads the data coming out of `c1` in parallel, stores it, and serializes it in an output port when ready. That module needs three clocks: `clk_c1`, `clk_s2` and `clk_proc`. It also has the `rst` port, as any other modules with synchronous processes. The input data is written in the `c1_din` input port, and its associated orientation is written in `c1_ori`. Data coming from different scale in C1 are written in parallel. `en_c1` is a input port having of side 8 – one pin per scale in `c1` –

that indicates which scale from `c1_din` is valid. Finally, a `retrieve` input port indicates that the following module is ready to get new data. Output data is written serially in `dout` output port, and a flag called `en_dout` indicates when data in `dout` is valid.

As shown in Figure 4.12, that module has four major components: two BRAM-based buffers that store the data and write it in `din` when ready, an instance of `c1_handler` which gets the input data and provides it along with the address where it should be written in the buffers, and finally a controller `ctrl` with two processes that takes care of the controlling signals. The reason why we need two buffers is that we use a *double buffering*: the data is first written into buffer A, then when all the required data has been written the next data is written into buffer B while we read that of buffer A, then buffer B is read while the data in buffer A is overwritten with new data, and so on. This allows to avoid problems related to concurrent accesses of the same resources.

When new data in `c1_din` is available, – that is when, at least one of `en_c1`'s bits is set – the writting process is launched. This process, which is synchronized on the high-frequency `clk_proc` clock, proceeds as follows: if `en_c1`'LSB is set, the corresponding data is read from `c1_din` and sent to `c1_handler` along with an unsigned integer identifying its scale. Then the second LSB of `en_c1` is read, and the same process is repeated until all 8 bits of `en_c1` are checked.
In parallel, `c1_handler` returns its input data along with the address where it should be written in BRAM. Both are sent to the buffer available for writing, which takes care of the writing of the data in its inner BRAM. Once data is ready, i.e when all C1 feature maps for an image are written in the buffer, then that buffer becomes read-only, as new incoming data is written in the other buffer. Every time the `retreive` input signal switches state, data is written into `dout` and `en_dout` is set. When the data is written, it is always by batches of four values, one per orientation.

`c1_handler` That module handles the pixels sent from the `c1_to_s2` and its corresponding scale, and simply rewrites it in its output ports with the address to which it should be written in `c1_to_s2` write buffer. Its input ports consist in `clk` which get the clock on which it should be synchronized, the `rst` port allowing to reset the component, the `din` port getting the C1 value to be handled, the scale of which is written in the `scale` input port, the `rst_cnts` that allows to reset all of this module's inner counters used to generate the address, and the `en_din` input port indicating when valid data is available and should be processed. This module's output port consist in `dout` which is used to rewrite input data, `addr` which indicates the address where to write the data in BRAM and `en_dout` indicating that output data is available.

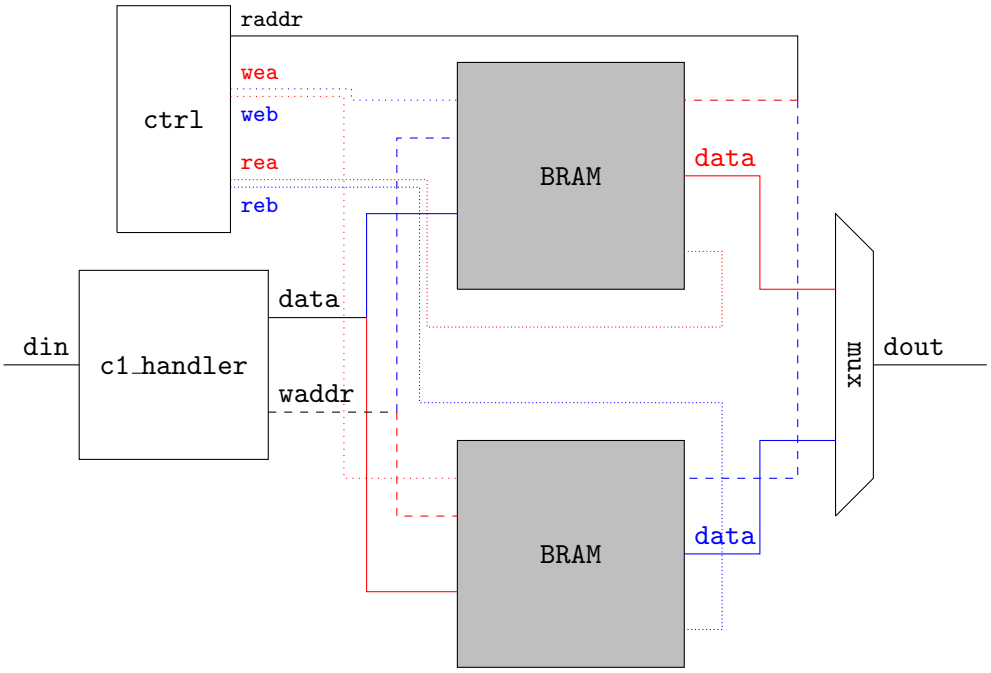


FIGURE 4.12: `c1_to_s2` module. The blue and red lines show the data flow in the two configurations of the double-buffering. The data goes through `c1_handler`, where the address to which it should be written is generated and written in `waddr`. The `rea` and `reb` signals control the enable mode of the BRAMs, while the `wea` and `web` enable and disable the write modes of the BRAMs. When the upper BRAM is in write mode, `wea` and `reb` are set and `web` and `rea` are unset. When the upper buffer is full, those signals are toggled so that we read the full buffer and write in the other one. Those signals are controled thanks to the `ctrl` component, which also generates the address from which the output data should be read from the BRAMs. Data read from both BRAMs are then multiplexed into the `dout` output port. Pins on the left of both BRAMs correspond to the same clock domain, and those on the right belong to another one so that it is synchronized with following modules.

That module works as follows. It has 8 independent counters, one per scale. Let c_s^n be the value hold on the counter associated to scale s at instant n. When `en_din` is set, assuming the value read from `scale` correspond to the scale s coded as an unsigned integer, the data read from `scale` is simply written in `dout` and the value written in `addr` is simply $c_s^n + o_s$, again coded as an unsigned integer, where o_s is an offset value as given in Table 4.5. Those offsets are determined so that each scale has its own address range, contiguous with each others, under the conditions given in Section 4.3.1:

$$o_0 = 0, \tag{4.24}$$

$$\forall s \in \mathbf{Z}^* \quad o_s = \sum_{k=0}^{s-1} 4{S_k}^2, \tag{4.25}$$

where S_k is the size of the C1 maps at scale k, also given in Table 4.5. Once all pixels have been handled by `c1_handler`, the module's counters must be reset by setting and then unsetting the `rst_cnts` input signal.

scale s	1	2	3	4	5	6	7	8
C1 patch side	31	24	20	17	15	13	11	10
offset o_s	0	3844	6148	7748	8904	9804	10480	10964

TABLE 4.5: Offsets used to computed addresses in `c1_to_s2` modules.

4.3.3 `s2c2`

That module gets as input the serialized data produced by `c1_to_s2`. and performs the operation required in HMAX's S2 and C2 layers. It has two main components, `s2` and `c2`, that respectively take care of the computations needed in HMAX's S2 and C2 layers. In order to save hardware resources, the pre-learnt in S2 filters are multiplexed as it is done in `s1`: every time new data arrives, pattern-matching are performed with some of the pre-learnt S2 patches in parallel, then the same operations are performed with other pre-learnt patches and same input data, and so on until all pre-learnt patches were used. We shall define here for latter use a *multiplexing factor* that we shall denote M_{S2C2}, which corresponds to the amount of serial computations required to perform computations on all S2 patches for a given input data. Is most useful output port is called rdy, and is connected to `c1_to_s2`'s `retreive` input port, to warn it when it is ready to get new data.

4.3.3.1 `s2`

This module handles the data coming out of `c1_to_s2` as well as the pre-learnt patches, matches those patterns and returns the results. Its input pins firstly consist in `clk` and `clk_proc` that each get a clock signal: the first one is the clock on which the input data is synchronized and the other one synchronizes the computations. It also has a `rst` input port allowing to reset it. The data should be written in the `din` port, and a port called `en_din` indicates that the input data is valid. After performing the pattern matching operations, the data is written into the `dout` output port, along with an identifier into the `id_out` output port. Finally, `en_new` allows the other module to be warned that new data is available, `en_dout` indicates precisely which parts of `dout` carry valid data and should be read and `rdy` indicates when the process is ready to read data from `c1_to_s2`.

That modules has three major components, which shall be described in the next Sections: `s2_input_manager`, which handles and organizes input data; `s2_coeffs_manager`, which handles and provides the coefficients of the pre-learnt filters; and `s2_processors`, which takes care of the actual pattern-matching operations.

Figure 4.13 shows the dataflow in that module. We shall now described in more details the its sub-modules `s2_coeffs_manager`, `s2_coeffs_manager` and `s2_processors`.

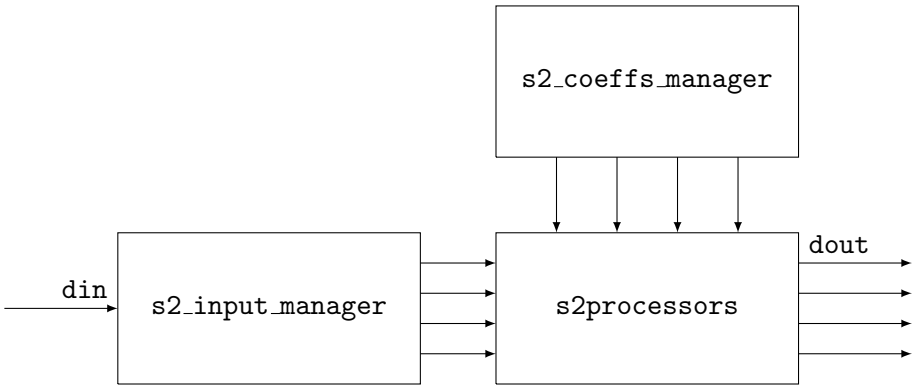


FIGURE 4.13: Dataflow in `s2c2`. The data arriving to the module is handled by `s2_input_manager`, which make it manageable for the `s2processors`. The latter also gets the pre-learnt filter needed for the pattern-matching operations from `s2_coeffs_manager` in parallel, and perform the computations. Once it is over, the data is sent in parallel to the `dout` output port, which feed the next processing module.

4.3.3.2 `s2_input_manager`

This module's purpose is somewhat similar to that of `s1`'s `pixman` module: managing the incoming data and reorganizing it in a way that makes it easier to process. It gets input data from `c1_to_s2` serially and provide a $N \times N \times 4$ map of C1 samples, where N is the side length of the available map. Its input ports gather a `clk` port the clock and a `rst` port allowing to reset the module, and also a `din` port where the data should be written and an `en_din` port that should be set when valid data is written into `din`. The output map may be read from the `dout` output port and its corresponding scale in C1 space is coded as an unsigned integer and written into the `dout_scale` output port. Finally, the `input_matsize` output gives a binary string w.r.t the value of the aforementioned N variable according to Table 4.6. Individually, each bit of `dout_scale` allow to enable and disable `s2bank` modules, which takes care of the actual pattern-matching operation and which are described in Section 4.3.3.6.

`s2_input_manager` mainly consists in two components: `s2_input_handler`, which get C1 samples serially as input and returns vertical stripes of those samples; and an instance of the `pixmat` component described in Section 4.3.2.1. However, `pixmat` is not used here in exactly the same way as in `s1`. First of all, we consider here that a "sample" stored in `pixmat` does not actually correspond to a single sample of a C1 map at a given location, but to an ensemble of four C1 samples, one per orientation. Furthermore, contrary to `s1`, the feature maps produced by `c1` do not have the same sizes, as stated earlier in Table 4.5. Thus, we instantiated a `pixmat` component adapted to the maximum size of C1 feature maps, i.e 31×31. The problem is that `pixmat`'s `en_dout` signal is only set when the whole matrix is ready, which make it impractical for C1 feature maps smaller than 31×31.

N	0	4	8	12	16
dout_scale	0000	0001	0011	0111	1111

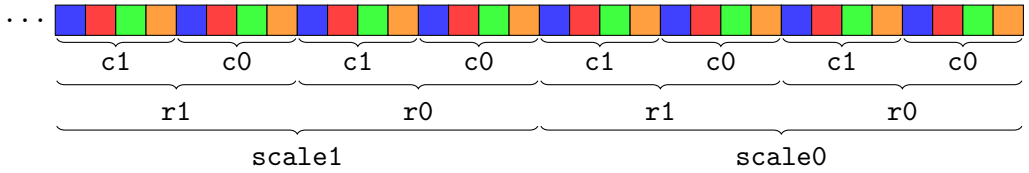
TABLE 4.6: Mapping between N and dout_scale.

To address that issue, we chose to ignore pixmat's en_dout port, and to use a state machine that shall keep track of the data in a similar way to that of pixmat, although it manages better the cases where the feature maps are smaller than 31×31: the process is similar but the line width depends on the scale to which the input data belongs. That scale is determined by an inner counter: knowing how many samples there are per scales in C1, it is easy to know the scale of the input data.

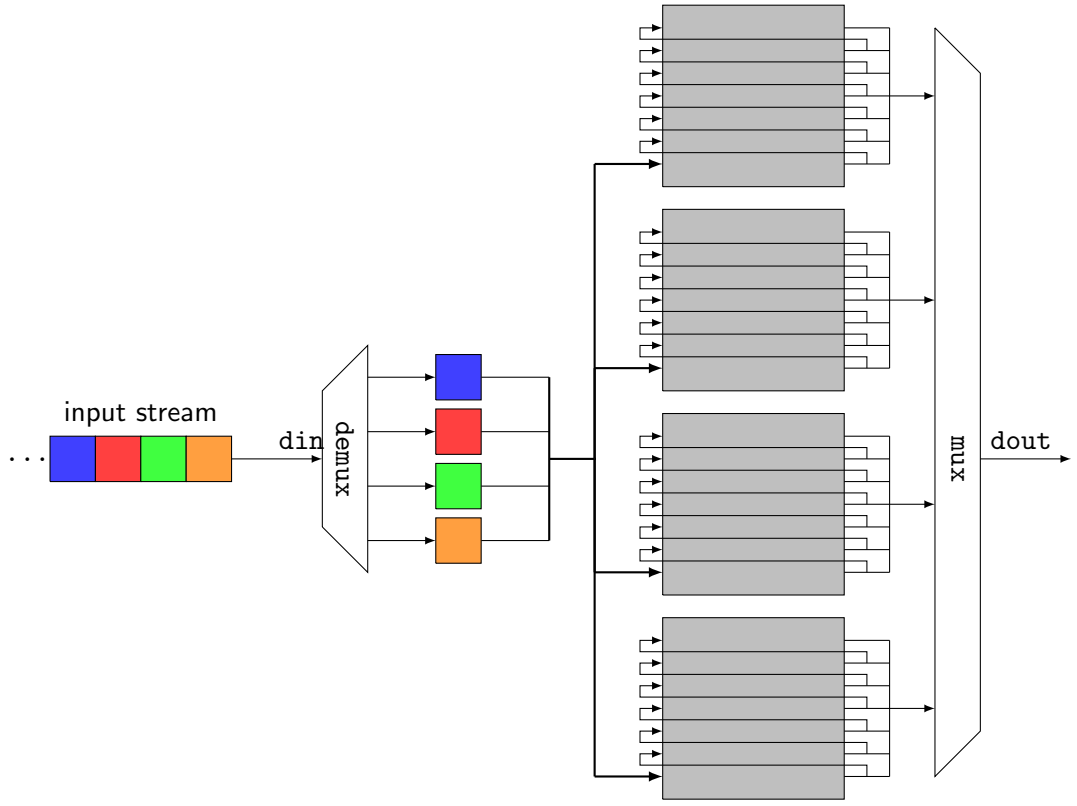
s2_input_handler The reorganization of the data arriving sample by sample in stripes that can feed pixmat is performed by that module. As a synchronous module, it has the required clk and rst input ports, the data is read from its din input port and valid data is signaled with the usual en_din input port. Output stripes are written in the dout output port, along with the identifier of their scales which are written in the dout_scale output port. Finally, the en_dout output port indicated that data from dout_scale is valid.

Let's keep track of the organization of the data that arrives in that module. Pixels arrive serially, as a stream. The first pixels to arrive are those of the C1 maps of the smallest scale. Inside that scale, the data is organized by rows, then columns, and then orientation as shown in Figure 4.14a. The first thing that module does is to demultiplex the orientations, so that every word contains the pixels of all of the orientations, at the same locations and scale. Once this is done, this new stream may be processed as we explain now.

As presented in Figure 4.14b, that module has 8 instances of the s2_pix_to_stripe component – one for each size of C1 feature maps – that produces the vertical stripes given input samples and generic parameters such as the desired stripe's height and width. Only one of those instances is used at a time, depending on the scale (which is computed internally depending on the amount of acquired samples). Thus, at scale 0 the C1 feature maps are 31×31 and the only active module is the 31×31 one. When processing samples of scale 2, which means 24×24 feature maps, the only active module is the 24×24 one, and so one. Whatever its side, the generated stripe is written in dout and its corresponding scale in dout_scale. Finally, en_dout indicates which data from dout is valid – this is somewhat redundant with dout_scale, but makes it easier to interface that model to the others.



(A) Organization of stream arriving in `s2_input_handler`. Each color indicates the orientation of the C1 feature map the corresponding sample comes from. We assume here that those feature maps are 2×2. `cX` indicates that the samples are located in the X-th column in their feature maps, and `rX` indicates that the samples are located in the X-th row.



(B) `s2_handler` module. Orientations are first demultiplexed, and written in parallel into the relevant `s2_pix_to_stripe`, shown here in gray. There is one `s2_pix_to_stripe` per scale in C1 feature maps – *i.e* 8. The output of those compinents are then routed to the `dout` output port, using a multiplexer.

FIGURE 4.14: Data management in `s2_handler`. Figure 4.14a shows how the arriving stream of data is organized. Figure 4.14b shows how this stream is processed.

`s2_pix_to_stripe` That module performs the actual low-level operations needed to transform a stream of serial samples into stripes. As any other synchronized modules, it has `clk` and `rst` ports, which are used respectively for synchronization and resetting module. The data arrive through an input port called `din`, and valid input data are signaled thanks to the `en_din` signal. The way the stripes are created is similar what is performed in `pix_to_stripe` – see Section 4.3.2.1. It is recalled here that the data is encoded using codebooks and boundaries determined with Lloyds algorithm as explained in Section 4.2.3, thanks to the `s1degrader` module.

4.3.3.3 `s2_coeffs_manager`

This modules takes care of the storage of the coefficients of the pre-learnt patches of S2, and retreive those that are required at a given time to provide them to the `s2processors` component, where the pattern-matching operations takes place. Its input port simply consist in a `clk` for the clock and an `addr` where the address of the desired coefficients is given. The coefficients are written in the `coeffs` output port, and their codebook is written in the `cb` output port.

At a given instant, many pattern-matching operations are performed in a Single-Instruction-Multiple-Data (SIMD) manner. Therefore many operands must be provided at each clock cycle. In order to simplify the process, all the needed data are retrieved at once, meaning that we need to store the data in several ROM accessible in parallel, as done in `s1`'s `coeffs_manager` – see Section 4.3.2.1. The output given by those ROMs contain both the coefficient of the pre-learnt patch in their Lloyd encoding and the corresponding codebooks. The data is then simply separated, rerouted, concatenated and written into `coeffs` and `cb`.

4.3.3.4 `s2processors`

That module takes care of the actual computations in S2. Pattern matching of all S2 units of all sizes is performed here. Data are synchronized on the *pixel clock* which is provided to this module via its `clk` input port. Operations, however, are synchronized on the *S2 process clock* of much higher frequency, given by the `clk_proc` input port. The module also has the compulsory `rst` clock allowing to initialize it. The data resulting from the processes of the previous layers is passed through the `din` input port, along with its codebook identifier via the `cb_din` input port. The pre-learnt patterns to be used for the pattern matching operations are passed through the `coeffs` input bus, and all their corresponding codebooks identifiers are given to the module via an input port called `cbs_coeffs`. Finally, the `id_in` input port gets an identifier that allows to keep

track of the data in the latter `c2` module, and the `en_din` allows to enable or disable the module.
Regarding the output ports, they consist in the `dout` port which provide the results of *all* the pattern matching operations performed in parallel, the `id_dout` port that simply gives back the identifier provided earlier via the `id_in` port, a "`rdy`" output port that warns that that module is ready to get new data, and finally an `en_dout` output bus that indicates which data made available by `dout` is valid; this is required due to the fact that, as we shall see, pattern matching operations are not performed at all positions of the input C1 maps, depending on the various sizes of the pre-learnt pattern. Thus, data are not always available at the same time, and we need to keep track of this.

For each size of the pre-learnt S2 patches, i.e $4\times4\times4$, $8\times8\times4$, $12\times12\times4$, $16\times16\times4$, this module implements two components: `s2bank` that performs the actual pattern matching operation, and `corner_cropper` that makes sure that only valid data is routed to the `s2bank` instance. Data arriving from `din` corresponds to a matrix of $16 \times 16 \times 4$ pixels: all of it is passed to the `s2bank` instance that match input data with $16\times16\times4$ patterns. The data fed to `s2bank` instances performing computations for smaller pre-learnt pattern corresponds to a chunk of the matrix cropped from the "corner" of the pixel matrix.

Figure 4.15 sums up the data flow in `s2processors`. We shall now describe the `corner_cropper` and `s2bank` modules.

4.3.3.5 corner_cropper

This module has a purpose similar to the `image_cropper` used in `s1`, except that, as explained in Section 4.3.2.1, `image_cropper` crops a *centered* fragment of its input pixel matrix. This module, however, crops the bottom-right corner of the input image. As in `image_cropper`, this is simply done by not connecting all inputs arriving from the `din` input port to the `dout` output port. This behaviour is shown in Figure 4.15.

4.3.3.6 s2bank

This module consists, as suggested in its name, in a bank of `s2unit` components (see below), all processing vectors of the same size. As `s2processor`, it needs the mandatory `clk` and `clk_proc` input ports to get the *pixel* and *process* clocks. The `rst` input pin allows to reset the module, and `en_din` allow to enable or disable the module. The data routed by the `corner_cropper` module is written in the `din` input port along with the corresponding codebook in `cb_din`. The `coeffs` input port gets the coefficients of

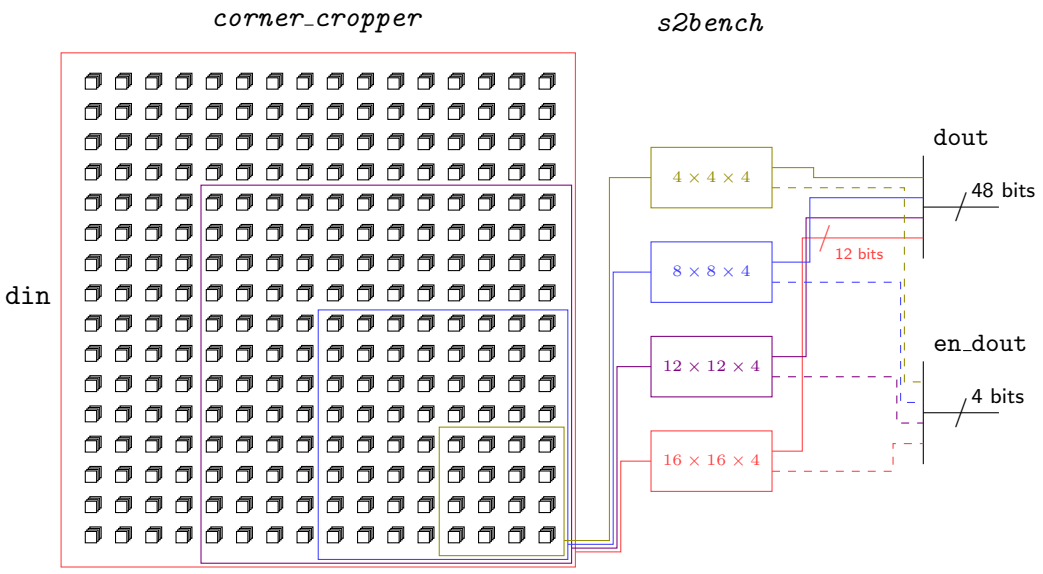


FIGURE 4.15: Data flow in `s2processors`. Names in italic represent the *components* instantiated in that module, and plain names show input and output ports. Only `din`, `dout` and `en_dout` are represented for readability. Each square in `din` represent one of the 1024 pixels read from `din`, and each set of four squares represents the pixels from C1 maps of the same scale and locations, and the four orientations. The `corner_cropper` module makes sure only the relevant data is routed to the following `s2bank` components. Those components perform their computations in parallel. When the data produced by one or several of those instances is ready, it is written in the corresponding pins of the `dout` output ports and the relevant pins of the `en_dout` output port are set.

the pre-learnt vector used for the pattern matching operation, and the corresponding codebook is got via the `cb_coeffs` input port.

4.3.3.7 `s2unit`

That module takes care of the computation of a single pattern matching operation in S2. As its top module `s2processors`, it has `clk` and `clk_proc` input ports that respectively get the data and system clocks. It also has `rst` input ports for reset. The operands consist on one hand in the data produced by the `s1c1` module and selected by `corner_cropper` and on the other hand in the pre-learnt pattern with which the Manhattan distance is to be computed. They are respectively given to that module via the `din` and `coeffs` input ports. The data arrive in parallel in the form of the optimized encoding described in Section 4.2, and as explained there this encoding requires a codebook. Since there is a codebook per C1 map, the identifiers of the codebooks required for the input data and the pre-learnt pattern are respectively given by the `cb_din` and `cb_coeffs` input ports. The identifier mentioned in `s2processors` passed by the `id_in` input port, and the module can be enabled or disabled thanks to the `en_din` input port. The Manhattan distance computed between the passed vectors is written to the `dout`

output port, along with the corresponding identifier which is written to the `id_out` output ports. Finally, an output port called `en_dout` indicates when valid data is available.

The Manhattan distance is computed here in a serial way, synchronized on the `clk_proc` clock. This computation is performed by a component called `cum_diff`, which shall now be described. Shift registers as described in Section 4.3.2.3 are also used to synchronize data.

`cum_diff` As suggested by its name, this module computed the absolute difference between two unsigned integers, and accumulates the result with those of the previous operations. To that end, it needs the usual `clk` and `rst` input ports for respectively synchronization and resetting purposes. It also needs two operands, which are provided by the `din1` and `din2` input ports. An input port called `new_flag` allows to reset the accumulation to 0 and start a fresh cumulative difference operation, and the `en_din` flag allows to enable computation. That module has a single output port called `dout`, which provides the result of the accumulation as it is computed. It is not required to have an output pin stating when the output data is valid, for the reason that the data is *always* valid. Knowing when the data actually correspond to a full Manhattan distance is actually performed in `s2unit`.

In this Section, we described principles of the `s2` module. Next Section does the same for the `c2` module.

4.3.4 `c2`

HMAX's final stage is performed in the `c2` module. It is synchronized, and thus has a `clk` input port expecting to get a clock signal, as well as an `rst` input port allowing to reset the component. The data used to performe the computation is obtained thanks to the `din` input port, and it arrives in parallel. The `id_in` input port allows to indicate which of the data from `din` are valid, and a `new_in` input port allows to warn about the arrival of new data. After performing of the maximum operations, the results for all pre-learnt vectors in S2 are written in parallel into the `dout` output port, and the last output port, which is called `new_out`, indicates that new data is available through `dout`. As done in the `c1_to_s2` presented in Section 4.3.2.5, we use a *double-buffering* design pattern to manage output data.

4.3.4.1 `c2_to_out`

This is the very final stage of our HMAX hardware implementation. It gets the data given by the `c2` module in parallel, and serialize it in a way very similar to that of the `c1_to_s2` module. Its input pins consist in the usual `clk` and `rst` respectively for synchronization and reset purposes, as well as a port called `din` that get the input data and `new_in` that indicates when new data is available. Serial output data is written in the `dout` output port, and the `en_dout` output port indicates when the data from `dout` is valid.

As in `c2_to_out`, the parallel data from `din` is simply read and written serially into the `dout` output port, while `en_dout` is set. When this is done, `en_dout` is unset again.

In this Section, we described the architecture of our VHDL model for the HMAX frameworks, taking into account our own optimizations along with other simplification from the literature. That implementation was purposely naive, in order to compare it with the state-of-the-art. Next Section focuses on the implementation results of that model on a hardware target.

4.4 Implementation results

In the previous Section, we described the architecture of our VHDL model. The next step is to synthesize and implement it for a particular device. We chose to target a Xilinx Artix-7 200T FPGA. Both synthesis and implementation were performed with Xilinx Vivado tools.

We first examine the utilization of hardware resources – in particular, we shall see that our model does not fit on a single device as is. We then study the timing constraint of our system, including the latency it induces.

4.4.1 Resource utilization

We synthesized and implemented our VHDL code using Xilinx's Vivado 2016.2, targeting a XC7A200TFBG484-1 platform. Results are shown in Table 4.7. On can see that there is still room for other processes on the FPGA, for instance of a classifier.

Now that we studied the feasability of the implementation of our model on hardware devices, let's study the throughput that it may achieve.

Resource	Estimation	Available	Utilization (%)
Look-up tables	58204	133800	43.50
Flip-flops	158161	267600	59.10
Inputs/outputs	33	285	11.58
Global buffers	6	32	18.75
Block RAM	254	365	69.59

TABLE 4.7: Resource utilization of HMAX implementation on XC7A200TFBG484-1 with the proposed simplifications. The proportion of used flips-flops is high enough to cause problems during implementation. However, the biggest issue comes from the fact that we use way too many blocks RAM for a single such target.

4.4.2 Timing

Our model works globally as a pipeline, where each module uses its own resources. Therefore, the overall time performances of the whole chain is determined by the module that takes longest. In order to evaluate how fast is our model in terms of frames per second, we shall now study, for each stage, the timing constraints it requires.

Let's begin with the S1 layer. The convolution is computed at 128×128 places of the input image. As detailed in Section 4.3.2.1, the sums of implied by the convolution are performed row-wise in parallel, and the results per row as then sum sequentially. Thus, for a $k \times k$ convolution kernel, k sums are of k elements are performed in parallel, and each one of them takes 1 cycle per element – hence, k cycles. That leads to k elements, which are them sum using the same strategy, and thus requiring another k cycles, thus totalizing $2k$ cycles. Since we use a 4-to-1 multiplexing strategy to compute the output of the orientations one after the other, all scales are processed in parallel and the biggest convolution kernel is 37×37, the convolution takes $128 \times 128 \times 8 \times 37 = 4.85 \times 10^6$ clock cycles to process a single 128×128 image.
As for the C1 layer, it processes the data as soon as it arrives, and thus no bottleneck is involved there.
The S2 layer is the most demanding in terms of computations. Computations are performed only when all the required data is here, in order to save as most time as possible, as explained in Section 4.3.3.5. Considering we use a 25-to-1 multiplexer to process all S2 filters, the time T_{S2} required by this stage is given by the time required by that layer may be written as

$$T_{S2} = 25\left(16 \times 16 \times 4M_{16} + 12 \times 12 \times 4M_{12} + 8 \times 8 \times 4M_8 + 4 \times 4 \times 4M_4\right), \quad (4.26)$$

where M_i is the number of valid $X_i \times X_i$ patches in the C1 feature maps where patches bigger than $X_i \times X_i$ with $X_i = 4i$ are not valid, and may be expressed as

$$M_i = N_i - N_{i+1} \quad (4.27)$$

where N_i is the number of valid $X_i \times X_i$ patch in the C1 feature maps. Hence, we have

$$\begin{aligned} T_{\text{S2}} =& 25\,[16 \times 16 \times 4N_{16} \\ &+ 12 \times 12 \times 4\,(N_{12} - N_{16}) \\ &+ 8 \times 8 \times 4\,(N_8 - N_{12}) \\ &+ 4 \times 4 \times 4\,(N_4 - N_8)] \qquad (4.28) \\ =& 2240N_{16} + 1600N_{12} + 960N_8 + 320N_4. \qquad (4.29) \end{aligned}$$

Let's now evaluate the N_i. Considering that some the C1 feature maps are smaller than some of the pre-learnt patches and that in such case, no computations are performed, we may write

$$N_i = \sum_{k=1}^{8} \max\left(\left\lfloor \frac{128}{\Delta_k} \right\rfloor - i + 1, 0\right)^2. \qquad (4.30)$$

with Δ_k defined in Table 2.1. Hence we have

$$\begin{aligned} N_{16} &= 435 \\ N_{12} &= 821 \\ N_8 &= 1437 \\ N_4 &= 2309, \end{aligned} \qquad (4.31)$$

which gives

$$T_{\text{S2}} = 2240 \times 435 + 1600 \times 821 + 960 \times 1437 + 320 \times 2309, \qquad (4.32)$$

and thus $T_{\text{S2}} = 110.16 \times 10^6$ clock cycles.
Finally, C2 processes the data as soon as it arrives in a pipelined manner, as done in C1. Hence, it doesn't bring any bottleneck.

We see from the above analysis that the stage that takes most time is S2, with 4.41×10^6 clock cycles per image. Assuming we have a system clock cycle of 100 MHz, we get **22.69 FPS**.

4.5 Discussion

One of the most interesting contributions about HMAX hardware implementation is the work of Orchard *et al.*, described in Section 4.1.1 – as mentioned in Section 2.2.2.1, there exists several implementations of either parts of the model or of the whole model on boards containing many FGPAs, but we shall focus here only on that work, as it is the

only one to our knowledge aiming to implement the whole model on a single FPGA. In that work, they implemented their algorithm on a Virtex 6 XC6VLX240T FPGA, while we targeted an Artix-7 XC7A200TFBG484-1 device. Table 4.8 sums up the resources of those two devices; we see that the Virtex-6 FPGA has slightly more resources than the Artix-7, however the two devices have roughly the same resources.

The strategy proposed in [99] is very different from what we proposed here. The huge computational gain they brought is largely due to the use of separable filters for S1, which allow to use very few resources as explained in Section 4.1.1.1. The fact that, in their implementation of S1, filters are multiplexed across scales instead of across orientations as we did here, also allows to begin computations in the the S2 layer as soon as data is ready, while in our case we chose to wait for all C1 features to be ready before starting computation, using a double-buffer to allow a pipelined process. In their case, the bottleneck is the S1 layer, which forces them to process a maximum of 190 images per second. However, that amount is 8.37 times bigger than the FPS we propose. This is due to the fact that, while reducing data encoding seem to provide performances similar to those obtained with full double-precision floating point values, it does not take full advantage of the symmetries underlined by Orchard *et al.* in [99].
As for the S2 layer, Orchard *et al* claimed that they used 640 multipliers in order to make the computation as parallel as possible – however it is not very clear in that paper how exactly those multipliers were split across filters, and the code is not available online – hence direct comparison with our architecture is not feasible. However, with their implementation of S2 they claim being able to process 193 128×128 images per seconds, while our implementation gives 22.69 images per second, although it uses much less resources. Finally, we did reduce the precision of the data going from S1 to S2, but the computation in S2 is still performed with data coded on 24 bits integer – this is due to the fact that we did not tested the model when degrading the precision at that stage. Future work shall address that issue, and we hope to reduce the precision to a single bit per word at that stage. Indeed, in that extreme scenario the computation of the Euclidean distance is equivalent to that of the Hamming distance, i.e. the number of different symbols between two words of same length. That kind of distance is much easier to compute than classical Euclidean or even Manhattan distance, be it on FPGA or CPU. The rational behind that idea is that single bit precisions were successfully used in other machine learning contexts [85, 170], and such an implementation would be highly profitable for implementation on highly constraint devices.

Resource	XC6VLX240T	Artix 7 200T
DSP	768	740
BRAM	416	365
Flip-flops	301440	269200
Look-up tables	150720	136400

TABLE 4.8: Hardware resources comparison between the Virtex-6 FPGA used in [99], and the Artix-7 200T we chose.

4.6 Conclusion

This Chapter was dedicated to the optimizations of the computations that take place in the HMAX model. The optimization strategy was to use simpler operations as well as coding the data on shorter words. After that study, a hardware implementation of the optimized model was proposed using the VHDL language, targeting an Artix 7 200T platform. Implementation results in terms of resource utilization and timing were given, as well as comparisons with a work chosen as a baseline.

We showed that the precision of the data in the early stages of the model could be dramatically reduced, while keeping acceptable accuracy: only the 2 most significant bits of the input image's pixels were kept, and the Gabor filters' coefficients were coded on a single bit, as was proposed in [167]. We also used the coding strategy proposed in the same paper, in order to reduce the bit width of the stored coefficients and their transfer from modules to modules. We also instantiated less patches in S2 as proposed by Yu and Slotine [46], and we proposed to use the Manhattan distance instead of the Euclidean distance as in the initial model [168]. Those optimizations made the overall accuracy of the model lose XXX points in precision for an image classification task based on 5 classes of the popular Caltech101 dataset, while dividing the complexity in the S2 stage by 5 and greatly reducing the required precision of the data, hence diminishing the memory print and the needed bandwidth for inter-module communication.

A hardware implementation of that optimized model was then proposed. We aim to that implementation to be as naive as possible, to see how those optimization compared with the implementation strategy proposed by Orchard *et al.* [99]. Their implementation was made so as to fully use the resources of the target device, and thus they claimed a throughput much higher than ours. However, our implementation uses much less resources than theirs, and our optimizations and theirs are fully compatible. A system implementing both of them would be of high interest in the fields of embedded systems for pattern recognition.

Future research shall aim to combine our optimizations with the implementation strategy proposed by Orchard *et al*, thus reducing even further the resource utilization of

that algorithm. Furthermore, we shall continue our efforts towards that objective, by addressing the computation in the S2 layers: at the moment, they are implemented as Manhattan distance – we aim to reduce the precision of the data during those pattern matching operation to a single bit. That way, Euclidean and Manhattan distances are reduced to the Hamming distance, much less complex to compute.

Chapter 5

Conclusion

In this thesis, we addressed the issue of optimizing a bio-inspired feature extraction framework for computer vision, with the aim of implementing it on a dedicate hardware architecture. Our goal is to propose an easily embeddable framework, generic enough to fit different applications. We chose to focus on efforts on HMAX, a computational model of the early stage of image processing in the mammal's cortex. Although that model may not be quite as popular as others, such as ConvNet for instance, it is interesting in that it is more generic and only requires little training, while frameworks such as ConvNet often require the design of a particular topology and a large amount of samples for training. HMAX is composed of 4 main stages, each computing features that are progressively more invariant that the one before, to translations and small deformations: the S1 stage uses Gabor filters to extract low-level features from the input image, the C1 stage uses a max-pooling strategy to provide a first level of translation and scale invariance, the S2 feature matches pre-learnt patches with the feature maps produced by C1 and the C2 provides full invariance to translation and scale thanks to its bag-of-word approach by keeping only the highest responses of S2. The only training that happens here is in S2, and it may be performed using simple training algorithms with few data.

First, we aimed to optimize HMIN, which is a version of HMAX with only the S1 and C1 layers, for two particular tasks: face detection, and pedestrian detection. Our optimization strategy consisted in removing the filters that we assumed were not necessary: for instance, in the case of face detection, the most prominent features lie in the eyes and mouth, which respond best to horizontal Gabor filters. Hence, we proposed to keep only such features in S1. Furthermore, most useful information are redundant from scales to scales, thus we reduced further the complexity of our system by summing all the remaining convolution kernels in S1, and we reduced it to a manageable size of 9×9 which allows it to process smaller images. Doing so helped us to greatly reduce the complexity

of the framework, while keeping its accuracy to an acceptable level. We validated our approach on the two aforementioned tasks, and we compared the performance of our framework with state-of-the art approaches, namely the Convolutional Face Finder and Viola-Jone's for the face detection task, and another implementation of ConvNet and the Histogram of Oriented Gradients for the pedestrian detection task.
For face detection applications, we concluded that, while the precision of our algorithm is significantly lower than that of state of the art systems, our system still works decently on a real life scenario, where images were extracted from a video. Furthermore, it presents the advantage of being generic: in order to adapt our model to another task one would simply need to update the weights of the filter in S1 so as to extract relevant features, while state of the art algorithm were either design specifically for the considered task or would require particular implementation for it.
However, our algorithm does not seem to perform to a sufficient level for the pedestrian detection task, and more efforts need to be made to that end. Indeed, while our simplifications allowed our system to be the most interesting in terms of complexity, they also brought a significant drop in terms of accuracy, although more tests need to be made for that use case as our results are not directly comparable to those of the state of the art.

We then went back to the full HMAX framework with all four layers, and we studied optimizations aiming to reduce the computation precision. Our main contribution is the use of as few as two bits to encode the input pixels, hence using only 4 gray levels instead of the usual 255. We also tested that optimization in combination with other optimizations from the literature: Gabor filters in S1 were reduced to simple additions and subtractions, the output in S1 were quantized using Lloyd's encoding method, allowing to find the optimal quantization given a dataset, we divided by 5 the number of pre-learnt patches in S2 and we replaced the complex computation of Gaussians in S2 with much simpler Manhattan distance. We showed that all those approximations allow to keep an acceptable accuracy compared to the original model.
We then implemented our own version on HMAX on a dedicated hardware, namely the Artix-7 200T FPGA from Xilinx, using the aforementioned optimizations. That implementation was purposely naive, in order to compare it with state of the art implementation. The precision reduction of the input pixels allows to greatly reduce the memory needed when handling the input pixels, and made the computation of the S1 feature map being done on narrower data. Furthermore, the replacement of the Gabor filter coefficients by simple additions and subtractions allowed us to encode that instruction on a single bit – "0" for subtraction and "1" for addition – instead of a full coefficient, using for instance a fixed or floating point representation. The data coming out of S1 is then encoding using the codebooks and partitions determined thanks to

Lloyd's method, hence allowing to pass only words of 2 bits to the C1 stage. As for the S2 layer, the influence of data precision on the performance was not yet evaluated by the time that document was written, and hence all data processed here used full precision: input data are coded on 12 bits, and output data on 24 bits.
The main limit of our implementation is that is does not use the symmetries of the Gabor filters. That technique was successfully used in the literature to propose a full HMAX implementation on a single FPGA, allong with different multiplexing scheme that allow a higher throughput. Indeed, our implementation – which is yet to be implemented and tested on a real device – may process 4.54 164×164 frames per second, while the authors of the state of the art solution claimed that it may process up to 193 128×128 frames per second. It must be emphasized however that our implementation uses much less hardware resources, and that our optimizations and theirs are fully compatible. Hence, future development shall mainly consist in merging the optimization they proposed with those that we used.

Let's now give answers to the question we stated at the beginning of that document. The first one was: How may neuromorphic descriptors be chosen appropriately and how may their complexity be reduced? As we saw, a possible solution is to find empirically the most promising features, and keeping only the filters that respond best to it. Furthermore, it is possible to merge the convolution filters that are sensible to similar features. That approach led us to a generic architecture for visual pattern recognition, and one would theoretically need to change only its weights to adapt it to new problems.
The second question that we stated was: How the data handled by those algorithms may be efficiently coded so as to reduce hardware resources? We show that full precision is not required to keep decent accuracy, and that we can acceptable results using even only a few bits to encode parameters and input data. We also showed that that technique may be successfully combined with other optimizations.

Given the fact that nowadays, the most widely used framework for visual pattern recognition is ConvNet, it may seem surprising that we chose to stick to HMAX. The main reason is that their most well known applications are meant to run on very powerful machines, while on the contrary we directed our research towards embedded systems. We also found the bio-inspiration paradigm promising, and we chose to push as far as possible our study of frameworks falling in that categories, in order to use them to their full potential. While our contribution in deriving an algorithm optimized for a given task does not provide an accuracy as impressive as the state of the art, we claim that the architecture of that framework is generic enough to be easily implementable on hardware, and that only the parameters would need to change to adapt it to another task. Furthermore, our implementation of the general-purpose HMAX algorithm on FPGA is

the basis of a future, more optimized and faster implementation on hardware, combining the presented optimizations which allowed to keep low hardware resource utilization low and those proposed in the literature, that take full advantage of the features of an FPGA. Combining those contributions may take several form: one can imagine using a full HMAX model with all four layers, but with a number of filters in S1 greatly reduced, thus leading to an implementation on FPGA using even less resources. Or, one can imagine directly implementing the framework proposed for face detection, i.e. without the S2 and C2 layers, with the optimizations that we proposed for the S1 and C1 layers. Doing so would produce a very tight framework, with a low memory print and a low complexity.

However, one may argue that frameworks such as ConvNet are nevertheless more accurate than HMAX in most use case scenarios, that frameworks such as Viola-Jones have strikingly low complexities, and that the genericity we claim to bring does not make it up for it. With that consideration, we claim that the study carried out in Chapter 3 and 4 may still apply to those frameworks. Indeed, as was done in the literature, if one trains a ConvNet having a topology similar to that of the CFF, where the feature maps of the second convolution stage ultimately produce a scalar each, one may see that the weight affected to that scalar if close to zero, and hence the corresponding convolutions responsible for that feature map may simply be removed; furthermore, for a given task it may be easy to identify the shape of a Gabor filter that would allow to grasp interesting features – then, one can either use Gabor filters as the first stage of a ConvNet, as was done in the past, or initialize the weights of some convolution kernels before training.
As for our hardware implementation of HMAX, most of the optimizations we proposed may be used for ConvNet as well. For instance, one could still chose to train a ConvNet on input images with pixels coded on less than 8 bits. Furthermore, after training one could also imagine to replace all positive weights with 1 and negative weights with -1, and remove weights close to 0 – given that the dynamics of the weights is not too far from the $[-1, 1]$ range. We also confirmed that using those techniques in combinations with other techniques from the literature, such as Lloyd's algorithm for inter-layer communication, are usable without dramatically altering the accuracy. Hence, our example of implementation is perfectly applicable to other situations, and goes way beyond the sole scope of HMAX.

To conclude, we would back the position that claims that bio-inspiration is often a good starting point and that it may open perspectives that were not explored until then, but that we should not fear to quickly move away from it. Indeed, humanity conquered the skies with machines only loosely connected to birds, and submarine depths with boats that share almost nothing with fishes. Computer vision boomed very recently thanks to frameworks that are indeed inspired by cognitive theories, but the implementations

of those theories in industrial systems is far from mimicking the brain. But all those systems, at some point, were inspired by nature – and while it is not always the most fundamental aspect, going back to that viewpoint and rediscovering why it inspired a technology may shed new lights on how to go further and deeper in their improvement.

Appendix A

RBF networks training

A.1 Overview

Radial Basis Function neural network (RBF) fall into the fields of generative models. As suggested in its name, after fitting a model to a training set, that type of models may be used to *generate* new data [24] similar to the real one. RBF are also considered as *kernel* models, in which the data is processed by so-called *kernel functions* before the actual classification; the goal is to represent the data in a new space, in which it is expected to be more easily linearly separable – particularly in the case when that new space is of larger dimensionality than the space of the input data. Other well-known kernel-based models are e.g SVM. Although those models may be used for both classification and regression tasks, we shall detail here its use for classification tasks only.

A short presentation of such models is proposed in Section 2.1.1.1, page 11 – the reader is invited to refer to that Section to get an overview of the motivations and the classification stage of those models. This Appendix in only dedicated to one of the many training procedure, which is the one we chose in all of our experiments involving an RBF net in Sections 3.1.3 and 3.2.3. It consists in two stages, the *clustering* stage and the *output layer optimization* stage. The first stage consists in reducing the amount of training samples by merging them into clusters, to which every sample to classify shall be compared – the aim of this reduction is to reduce the computational cost of the classification stage. The second stage consists in fitting the output layer weights – in this case this is performed with a MSE minimization.

A.2 Clustering

This stage consists in reducing the training set to a more manageable size. The method we chose is based on the work of Musavi *et al.*, but a bit simpler as we shall see. It consists in merging neighboring vectors of the same categories into *clusters*, each represented in the network by a *kernel function* that is constituted of *center*, i.e a representation in the same space of one or several data points from the training set, and a *radius*, showing the generalization relevancy of that center: the bigger the radius, the better the center represents the dataset. As we shall see, this method allows to build highly non-linear boundaries between classes.

Let $X^1 = \left\{x_1^1, x_2^1, \ldots, x_N^1\right\}$ be the training set composed of the N vectors $\left\{x_1^1, x_2^1, \ldots, x_N^1\right\}$, and $T^1 = \left\{t_1^1, t_2^1, \ldots, t_N^1\right\}$ be their respective labels. As for many training algorithm, it is important that the x_i^1 are randomized, so that we avoid the case where all vectors of a category have neighboring indexes i. Let also $d\left(a, b\right)$ denote the distance between the a and b vectors. Although any distance could be used, we focus here on a typical Euclidean distance so that

$$d\left(a, b\right) = \sqrt{\sum_{i=1}^{M} \left(a_i - b_i\right)^2} \tag{A.1}$$

where a and b have M dimensions.
The clustering algorithm proceeds as follows [174]:

1. map each element x_i^1 of X^1 to a cluster $c_i^1 \in C^1$, the radius r_i^1 of which is set to 0,
2. select the first cluster c_1^1 from C^1,
3. select a cluster c at random from the ensemble C of the other clusters of the same class – let x be its assigned vector and r its radius,
4. merge the two clusters into a new one c_1^2, the vector x_1^2 of which is the centroid of c_1^1 and c:
$$\forall i \in \left\{1, 2, \ldots, M\right\} \quad x_{1i}^2 = \frac{x_{1i}^1 + x_i}{2}, \tag{A.2}$$
5. compute the distance d_{opp} between c_1^2 and the closest cluster $\hat{c} \in C^1$ of another category,
6. compute the radius r_1^2 of the new cluster c_1^2, as the distance between the new center x_1^2 and the furthest point of the new cluster:
$$r_1^2 = \frac{1}{2} d\left(x_1^1, x\right) + \max\left(r_1^1, r\right), \tag{A.3}$$

7. if $d_{\text{opp}} > \mu R$, where μ is a strictly positive constant, accept the merge and go back to 3 using $C \setminus \{c\}$ instead of C; if $d_{\text{opp}} \leq \mu R$, reject the merge and go back to 3 selecting another cluster,

8. repeat steps 3 to 7 until all clusters from C were considered, which leads to a new set of clusters C^2,

9. repeat steps 2 to 8 using C^2 instead of C^1 and $c_1^2 \in C^2$ instead of c_1^1, and continue using C^3, C^4 and so on until no further merge is possible.

A.3 Output layer training

As for the final layer, it is trained using a simple least mean square approach. Denoting W the weight matrix and T the matrix of target vectors, it can be shown [24] that we have

$$W = \left(\Phi^{\mathrm{T}}\Phi\right)^{-1}\Phi T \tag{A.4}$$

with

$$\Phi = \begin{pmatrix} \Phi_0(x_1) & \Phi_1(x_1) & \dots & \Phi_{M-1}(x_1) \\ \Phi_0(x_2) & \Phi_1(x_2) & \dots & \Phi_{M-1}(x_2) \\ \vdots & & \ddots & \\ \Phi_0(x_N) & \Phi_1(X_N) & \dots & \Phi_{M-1}(X_N) \end{pmatrix} \tag{A.5}$$

where Φ_i is the function corresponding to the i-th kernel, and where each vector of T has components equal to -1, except for it i-th component which is $+1$ if the categories of the vector it corresponds to is i.

Appendix B

Résumé en français

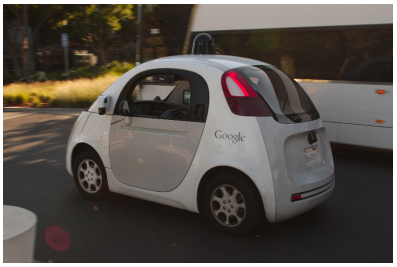
B.1 Introduction générale

L'automatisation est un processus visant à remplacer les opérateurs humains par des machines, afin de gagner en efficacité et de diminuer les coûts. Cela a été grandement appliqué au cas des tâches non qualifiées, mais les industries sont de plus en plus demandeuses de systèmes dit intelligents, capable d'analyser une information issue de leur environnement et de prendre une décision en conséquence. Ce genre de technologie est particulièrement utile par exemple dans le cas de la vidéo-surveillance, pour permettre au système de détecter automatiquement la présence d'un intrus, et si besoin de s'assurer qu'il s'agit bien d'une personne avant de lever une alarme. Cela peut également servir pour du contrôle de qualité sur une chaîne de production industrielle, notamment pour détecter des défauts sur les pièces produites. Des exemples de ces applications sont donnés en Figure B.1.

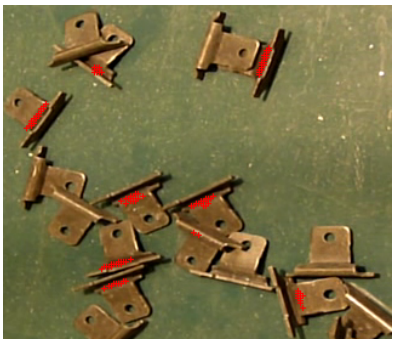
Afin de résoudre ce genre de problématiques, deux approches sont possibles : il est possible soit de créer soit-même un algorithme permettant d'analyser un signal et de prendre une décision, soit de produire un algorithme permettant à la machine d'*apprendre* par elle-même à résoudre la tâche, à partir d'exemples. Cette dernière approche fait appel à l'*apprentissage automatique*, qui est une branche faisant partie du domaine de l'intelligence artificielle.

C'est dans ce cadre que le projet NeuroDSP [5] a été lancé. Son but est de produire un processeur embarquable dans des systèmes plus vastes, afin de pouvoir analyser des signaux et prendre des décisions en conséquence au plus près du capteur. Ce processeur devra donc répondre à de fortes contraintes en termes de consommation d'énergie et

[1] Travaux de Michael Shick - Production peronnelle, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=44405988.



(A) Voiture autonome de Google[1].



(B) Contrôle qualité.

(C) Sécurité.

(D) Domotique.

FIGURE B.1: Exemples d'applications.

FIGURE B.2: Architecture NeuroDSP [5].

d'encombrement. Il est constitué de 32 blocs appelé P-Neuro, qui consistent chacun en 32 processeurs élémentaires (PE), pour un total de 1024 PE. Chacun de ces PE peut être vu comme un neurone d'un réseau de neurones artificiel, tel que le Perceptron. Au sein d'un P-Neuro, tous les PE exécutent la même opération sur des données différentes, constituant ainsi une architecture de type SIMD (*Single Instruction Multiple Data*), parfaitement adaptée aux calculs parallèle tels que nécessités dans les réseaux de neurones artificiels. Cette architecture est présenté en Figure B.2. Les travaux présentés dans ce documents ont été réalisés dans le cadre de ce projet.

Dans ce résumé, nous ferons tout d'abord un état de l'art de la littérature concernant ce domaine – nous y verrons les principales méthodes d'apprentissage automatique, leurs implantations sur matériel, et nous poserons les problématiques auxquelles nous répondront dans la suite du document. Une Section sera ensuite consacrée à

notre méthode de sélection de caractéristiques pour la classifications d'objets visuels. Nous présenterons ensuite une implantation optimisée d'un algorithme de classification d'images sur une plateforme matérielle reconfigurable. Finalement, la dernière Section présentera la conclusion de nos travaux.

B.2 État de l'art

Cette Section propose une brève revue de la littérature concernant les travaux présentés ici. Nous commencerons par les fondements théoriques de l'apprentissage automatique et de l'extraction de caractéristiques d'un signal. Nous verrons ensuite les implémentations matérielles existances pour ces méthodes. Finalement, nous proposerons une discussion au cours de laquelle nous établirons les problématiques auxquelles nous répondront dans ce documents.

B.2.1 Fondements théoriques

B.2.1.1 Méthodes de classification

Il existe de nombreuses approches permettant à une machine d'apprendre d'elle-même à classifier des motifs. Nous allons ici revoir les principales. Une approche extrêmement simple consiste à considérer l'intégralité des vecteurs dont nous disposons *a priori*, que l'on appelle *base d'apprentissage*. Lors de la classification d'un vecteur inconnu, on évalue une distance (par exemple, Euclidienne) avec tous les vecteurs de la base d'apprentissage, et on ne considère que les K plus proches. Chacun de ces vecteurs vote alors pour sa propre catégorie, et la catégorie ayant obtenue le plus de vote est retenue. On considère alors que le vecteur inconnue appartient à cette catégorie. Cette approche s'appelle KNN [6], pour *K-Nearest Neighbors*, et présente l'avantange d'être extrêmement simple à implanter. Cependant, lorsque le nombre d'exemples de la base d'apprentissage devient important ou que la taille des vecteurs devient trop grande, cette méthode devient trop complexe et trop consommatrice en mémoire pour être efficace, en particulier dans un contexte embarqué.

Il existe beaucoup d'autres méthode de classification de motifs, parmi lesquelles figurent en particulier les réseaux de neurones (cf. le Perceptron en Section B.1), ou des approches plus statistiques telles que les Machines à Vecteurs de Supports, ou SVM[2]. Les réseaux de neurones sont récemment devenus extrêmement populaires, depuis leurs utilisations par les entreprises Facebook et Google notamment pour leurs applications

[2] *Support Vector Machine*

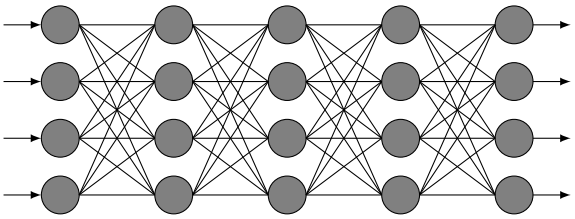


FIGURE B.3: Architecture *feedforward*.

de reconnaissances d'images. Nous nous intéresserons ici uniquement aux architectures dites *feedforward*, dans lesquelles les neurones sont organisées par couches et chaque unité transmet l'information à des neurones de la couche suivante – ainsi, l'information se propage dans un seul sens. Ce genre d'architecture est représenté en Figure B.3. Les connexions entre les unités sont appelés *synapses*, et à chacune d'entre elles est affecté un *poid synaptique*. Ainsi, la valeur d'entrée z d'un neurone de N entrée ayant des poids synaptiques $w_1, w_2, \ldots, w_N$ est donnée par

$$z = w_0 + \sum_{i=1}^{N} w_i x_i, \tag{B.1}$$

avec x_i les valeurs propagées par les unités de la couche précédente et w_0 un *biais*, nécessaire pour des raisons mathématiques. Une fonction non-linéaire, appelée *fonction d'activation*, est ensuite appliquée à z, et le résultat est propagé aux neurones de la couche suivante. Apprendre un réseau de neurones de ce type à éxecuter une tâche consiste à trouver les bons poids synaptiques, au moyen d'un *algorithme d'apprentissage*. Dans le cas des réseaux de neurones *feedforward* à plusieurs couches, l'algorithme le plus utilisé en raison de son efficacité et de sa faible complexité algorithmique est la descente de gradient stochastique – en effet, celui-ci peut être facilement exécuté au moyen d'une technique appelée *rétro-propagation de l'erreur*, qui permet d'evaluer rapidement la dérivée de la fonction de coût à optimiser [18, 19].

Une autre méthode de classification que nous utilisons dans ces travaux s'appelle le RBF, qui fait partie des méthode dites à noyaux. Elles consistent à évaluer un ensemble de fonction à base radiale au point représenté par le vecteur à classifier, et le valeurs produites par ces fonctions forment un nouveau vecteur qui sera classifié par un classificateur linéaire – e.g, un Perceptron. En revanche, dans ce cas la technique d'apprentissage utilisée est simplement une recherche de moindres carrés.

B.2.1.2 Méthodes d'extraction de caractéristiques

Afin de faciliter la tâche du classificateur, il est possible de faire appel à un algorithme d'extraction de caractéristiques, dont l'objet est de transformer le signal à classifier,

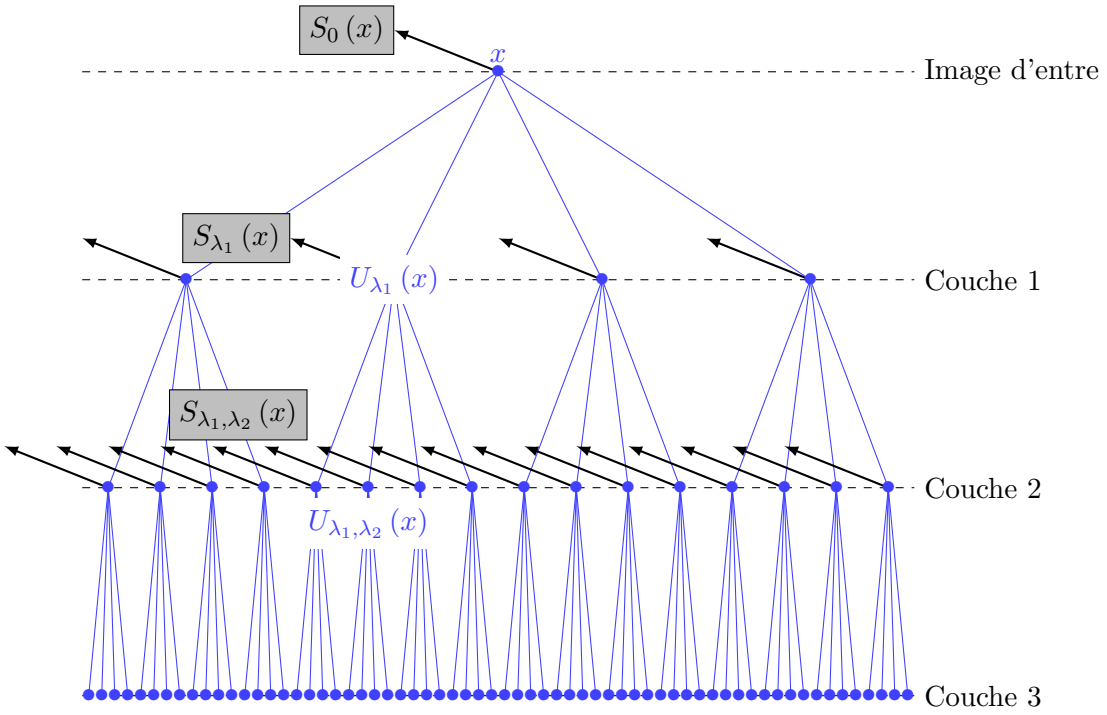


FIGURE B.4: Invariant scattering convolution network [38]. Chaque couche applique une décomposition en ondelette U_λ à l'entrée, et envoie le résultat auxquels a été appliqué un filtre passe-bas et un sous-échantillonage à la couche suivante. Les *scattering coefficients* $S_\lambda(x)$ ainsi produits forment le vecteur caractéristique à classifier.

afin d'en avoir une représentation invariante aux paramètres non pertinents. Par exemple, si la tâche est de détecter la présence d'un visage dans une image sans chercher à le localiser, et ce quelque soit sa position dans l'image, il serait souhaitable que cette représentation soit invariante aux translations. Pour parvenir à ce genre de résultat, plusieurs approches existent: il est possible de faire appel à des techniques de traitements de signal classiques, par exemple la transformée de Fourier ou la transformée en ondelettes. Mais il existe d'autres approches, plus complexes mais souvent plus efficaces. Nous pouvons par exemple citer le HOG (*Histogram of Oriented Gradients*) [36], ou encore *Scattering Transform*, qui consiste en une décomposition en ondelette d'ordre croissant suivit par des filtres passe-bas et un sous-échantillonage – une architecture typique de ce genre de méthode est donnée en Figure B.4.

Une autre approche possible est celle dite *bio-inspirée*. Elle consiste à utiliser des modèle computationnel de processus biologiques ou cognitifs pour répondre à des problèmes concrets. Parmis ces modèles, l'un des plus connue s'appelle HMAX [168] – il s'agit d'un modèle computationel des premières étapes du traitement des images par le cortex visuel des mamifères. Il consiste en 4 couches successives: S1, C1, S2 et C2. La couche S1 est constituée d'un banc de filtres de Gabor, permettant d'extraire des caractéristiques bas niveau selon différentes orientations et échelles. Un filtre de Gabor consiste en une

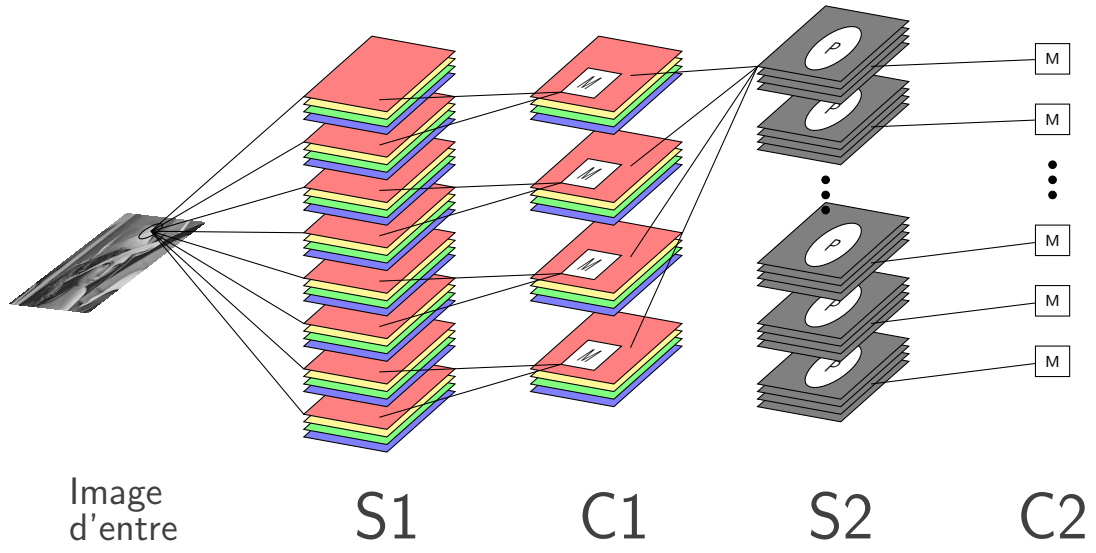


FIGURE B.5: HMAX [31].

gaussienne modulée par un cosinus, et peut être formalisé de la manière suivante:

$$G(x,y) = \exp\left(-\frac{x_0^2+\gamma^2 y_0^2}{2\sigma^2}\right) \times \cos\left(\frac{2\pi}{\lambda}x_0\right), \tag{B.2}$$

$$x_0 = x\cos\theta + y\sin\theta \quad \text{and} \quad y_0 = -x\sin\theta + y\cos\theta, \tag{B.3}$$

où γ est le ratio d'aspect, λ la longueur d'onde du cosinus, θ l'orientation du filtre et σ l'écart-type de la gaussienne. S1 comporte des filtres de 16 échelles et 4 orientations différentes, totalisant ainsi 64 filtres. Les paramètres des filtres sont donnés en Table B.1. La couche C1 fourni un premier niveau d'invariance aux translations et à l'échelle grâce à un ensemble de filtres maximum, dont la taille de la fenêtre N_k et le recouvrement Δ_k dépendent de l'échelle considérée, et sont donnés en Table B.1. La troisième couche, S2, compare les sorties de la couche C1 avec un ensemble de motifs pré-appris aux moyens de fonctions à base radiale. Finalement, la dernière couche C2 ne conserve, pour chacun de ces motifs pré-appris, que la réponse maximale, formant ainsi le vecteur caractéristique. Cet algorithme est présenté en Figure B.5.

D'autres méthodes d'extractions de caractéristiques ou de classifications (ou les deux), tels que SIFT [32], SURF [33] ou Viola-Jones [136] ont également connu une certaine popularité.

Enfin, il n'est pas possible de ne pas mentionner les réseaux de neurones à convolutions [49], qui sont les principaux contributeurs au succès que rencontrent les réseaux de neurones à l'heure actuelle. Leur approche est très simple: plutôt que de séparer l'extraction de caractéristiques de la classification, ces méthodes considèrent l'ensemble de la chaîne algorithmique et réalisent l'apprentissage sur son intégralité. L'extraction de

Couche C1			Couche S1		
Échelle	Taille du filtre maximum ($N_k \times N_k$)	Recouvrement Δ_k	Taille de filtre S1 k	Gabor σ	Gabor λ
Band 1	8×8	4	7×7 9×9	2.8 3.6	3.5 4.6
Band 2	10×10	5	11×11 13×13	4.5 5.4	5.6 6.8
Band 3	12×12	6	15×15 17×17	6.3 7.3	7.9 9.1
Band 4	14×14	7	19×19 21×21	8.2 9.2	10.3 11.5
Band 5	16×16	8	23×23 25×25	10.2 11.3	12.7 14.1
Band 6	18×18	9	27×27 29×29	12.3 13.4	15.4 16.8
Band 7	20×20	10	31×31 33×33	14.6 15.8	18.2 19.7
Band 8	22×22	11	35×35 37×37	17.0 18.2	21.2 22.8

TABLE B.1: Paramètres des couches S1 et C1 de HMAX [31].

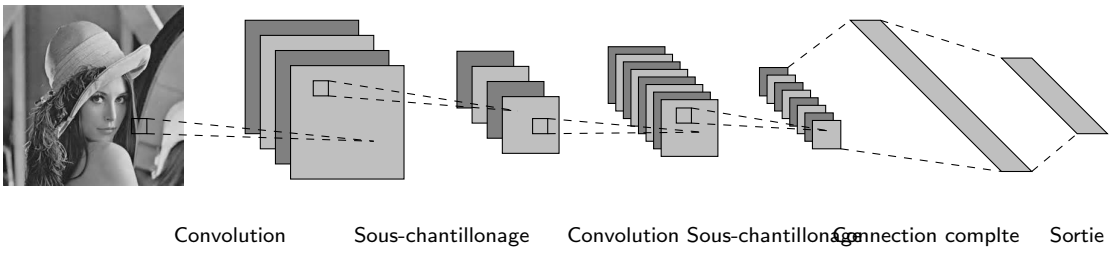


FIGURE B.6: Réseaux de neurones à convolutions [48].

caractéristiques consiste ici en une succession de convolutions et de sous-échantillonages. La classification, quant à elle, est réalisée au moyen d'un perceptron à plusieurs couches. À la différence de HMAX, les paramètres des convolutions de l'extracteur de caractéristiques sont déterminés à l'apprentissage. Cependant, cette souplesse est également le principal défaut de ce genre d'algorithmes : ils requièrent des bases d'apprentissage colossale pour optimiser un si grand nombre de paramètres. Cette méthode est illustrée en Figure B.6.

B.2.2 Implantations matérielles

Afin de répondre aux problématiques de l'embarqué, de nombreuses implantations matérielles de classificateurs, d'extracteurs de caractéristiques et même de réseaux de neurones à

TABLE B.2: Comparaison des principaux extracteurs de caractéristiques.

Méthode	**Précision**	**Apprentissage requis**	**Complexité**
Scattering Transform	Haute	Non	Élevée
HMAX	High	Oui, requière peu de données	Élevée
HOG	Raisonnable	Non	Basse
SIFT	Raisonnable	Non	Basse
SURF	Raisonnable	Non	Très basse
Réseaux de neurones à convolutions	Très élevée	Oui, requière beaucoup de données	Élevée

convolutions ont été réalisées. Il existe également de nombreuse implantations logicielles, mais nous ne les mentionneront pas dans ce résumé. HMAX lui-même a été implanté de nombreuses fois sur du matériel reconfigurable (FPGA) [91–98] – récemment, l'implantation la plus prometteuse pour ce modèle est celle proposée par [99]. Des travaux ont été menés en ce sens également pour les réseaux de neurones à convolutions [103, 107].

B.2.3 Discussion

Notre but est de proposer un système embarquable et générique de reconnaissance de motifs. Pour cela, nous allons choisir un extracteur de caractéristiques qui servira de base à nos futurs travaux. Le problème de la classification ne sera pas traité ici. La Table B.2 présente une comparaison des principaux descripteurs. Au vu de cette comparaison, nous avons décidé de porter notre étude sur HMAX, qui nous assurera de plus une certaine généricité.

Notre but est d'adapter cet algorithme à différentes tâches tout en conservant une généricité au niveau de l'architecture, et d'optimiser, notamment en termes de codage, ces algorithmes pour faciliter leur portage sur des cibles matérielles, ce qui amène les problématiques suivantes auxquelles nous nous efforcerons de répondre :

- Comment choisir des caractéristiques bio-inspirées de manière appropriée et comment réduire leurs complexités algorithmes ?
- Comment les données manipulées par ces algorithmes peuvent-elles être codées efficacement de façon à réduire l'utilisation des resources matérielles ?

Cette Section était dédiée à une revue de l'état de l'art lié à nos travaux. Dans la prochaine Section, nous allons répondre à la première problématique en décrivant notre contribution sur la sélection de caractéristiques. Dans la Section suivante, nous

détaillerons les optimisations réalisées sur HMAX en vue d'une implantation sur matériel. Enfin, la dernière Section sera consacrée aux discussion et conclusions de ces travaux.

B.3 Sélection de caractéristiques

Dans cette Section, nous allons présenter nos travaux concernant la sélection de caractéristiques en vue d'optimiser un algorithme, pour deux tâches précises: la détection de visages, et la détection de piétons.

B.3.1 Détection de visages

Nous allons tout d'abord présenter deux algorithmes populaires pour la détections de visages: Viola-Jones et le *Convolutional Face Finder* (CFF).

B.3.1.1 Viola-Jones

Viola-Jones [136] est un algorithme basé sur une cascade de classificateurs opérant sur des caractéristiques proches des ondelettes de Haar, représentées en Figure B.7. L'avantage de ces caractéristiques est que, quelque soit leurs tailles, elle peuvent être évaluées en un nombre constant d'additions et de soustractions, en utilisant une représentation particulière de l'image d'entrée appelée *image intégrale*. Dans cette représentation, le pixel situé au point de coordonnées (X, Y) a pour valeur

$$II\left(X,Y\right)=\sum_{x=1}^{X}\sum_{y=1}^{Y}f\left(x,y\right) \tag{B.4}$$

avec $f\left(x,y\right)$ la valeur du pixel situé au point de coordonnées (x, y) dans l'image originale. Cette représentation est illustrée en Figure B.8. Le calcule de l'image intégrale peut être réalisée itérativement:

$$F\left(x,y\right)=F\left(x-1,y\right)+s\left(x,y\right), \tag{B.5}$$

avec

$$s\left(x,y\right)=s\left(x,y-1\right)+f\left(x,y\right). \tag{B.6}$$

Intéressons-nous maintenant au calcul des caractéristiques. Prenons la caractéristiques présentées à la gauche de la Figure 3.1. En considérant l'image intégrale II, la réponse $r_l\left(x_1,y_1,x_2,y_2\right)$ à ce filtre est donnée par

$$r_l\left(x_1,y_1,x_2,y_2\right)=F\left(x_1,y_g,x_2,y_2\right)-F\left(x_1,y_1,x_2,y_g\right) \tag{B.7}$$

FIGURE B.7: Examples de caractéristiques utilisés dans Viola-Jones [30, 136].

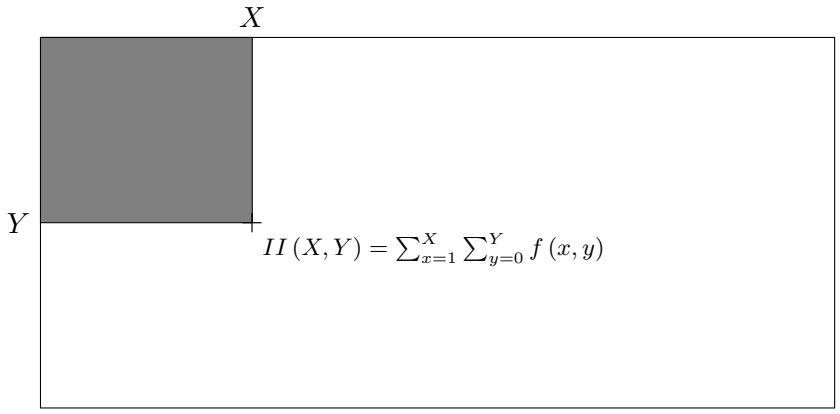


FIGURE B.8: Représentation en image intégrale [30].

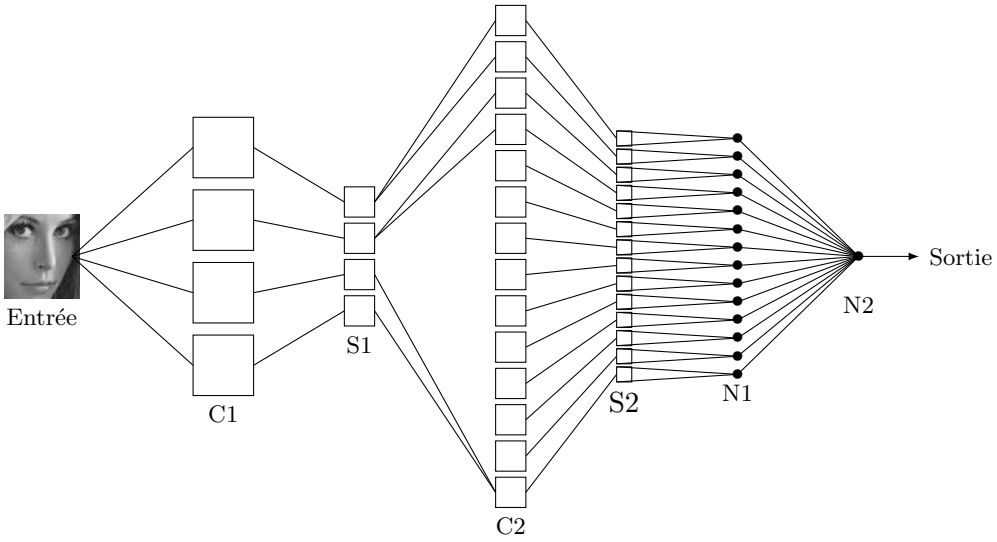


FIGURE B.9: *Convolutional Face Finder* [50].

avec $F(x_1, y_1, x_2, y_2)$ l'intégrale des pixels de l'image originale situés dans le rectangle délimité par (x_1, y_1) et (x_2, y_2).

B.3.1.2 CFF

Le CFF est une implantation particulière d'un réseau de neurones à convolutions, optimisée pour la détection de visages. Son architecture est données en Figure B.9.

Descripteur	HMIN	$\text{HMIN}_{\theta=\pi/2}$	$\text{HMIN}^{R}_{\theta=\pi/2}$
Précision (%)	95.78 ± 0.97	90.81 ± 1.10	90.05 ± 0.98

TABLE B.3: Précision des différentes versions de HMIN sur la base de données LFW_crop.

B.3.1.3 HMIN et optimisations

D'après l'article de Serre *et al.* [31], nous savons que pour détecter et localiser un objet dans une scène, il est préférable de n'utiliser que les deux premières couches de HMAX, i.e S1 et C1. Afin de voir quelles caractéristiques sont les plus pertinentes, et donc quelles caractéristiques peuvent être enlevées sans trop impacter la précision du système, nous avons observé les réponses des différents filtres de Gabor pour les visages. Les résultats sont montrés en Figure B.10. Nous pouvons y voir que les informations qui semblent les plus pertinentes sont celles correspondant à l'orientation $\theta = \pi/2$. Par ailleurs, nous pouvons voir que les informations sont semblables d'une échelle à l'autre. Ainsi, nous proposons de ne conserver que les filtres d'orientations $\theta = \pi/2$, et de les sommer, afin de n'avoir plus qu'une convolution. L'aspect du noyau de cette convolution est donné en Figure B.11. La sortie obtenue pour un visage après filtrage par ce noyau de convolution est donné en Figure B.12. Pour C1, la taille de la fenêtre du filtrage est $\Delta_k = 8$. Cet extracteur de caractéristiques sera appelé $\text{HMIN}_{\theta=\pi/2}$ dans la suite du document. Nous proposons ensuite de réduire la taille de ce noyau de convolution, qui comporte à l'heure actuelle 37×37 éléments, en le réduisant à 9×9 en utilisant une interpolation bilinéaire, ce qui lui permet de traiter des images 4 fois plus petites. Cette version du descripteur sera appelée $\text{HMIN}^{R}_{\theta=\pi/2}$.

B.3.1.4 Expérimentations

Afin de valider que nos optimisations n'impacte pas trop négativement la précision, nous avons mené une série d'expériences. Tout d'abord, nous avons testé HMIN, $\text{HMIN}_{\theta=\pi/2}$ et $\text{HMIN}^{R}_{\theta=\pi/2}$ sur la base de données de visage LFW_crop [141], où il s'agit de classifier chaque image entre deux catégories: *visage* et *non-visage*. Les résultats sont présentés en Table B.3 et en Figure B.13. Le classificateur choisi est RBF.

Un autre test, cette fois-ci de détection, a été mené sur la base CMU [143]. Les résultats sont montrés en Figure B.14. Ce test nous permet de comparer les performances de notre algorithme avec la littérature – les résultats sont présentés en Table B.4.

Dans cette Section, nous avons étudié comment optimiser HMIN pour la détection de visages. La prochaine Section est dédiée à une démarche similaire pour la détection de piétons.

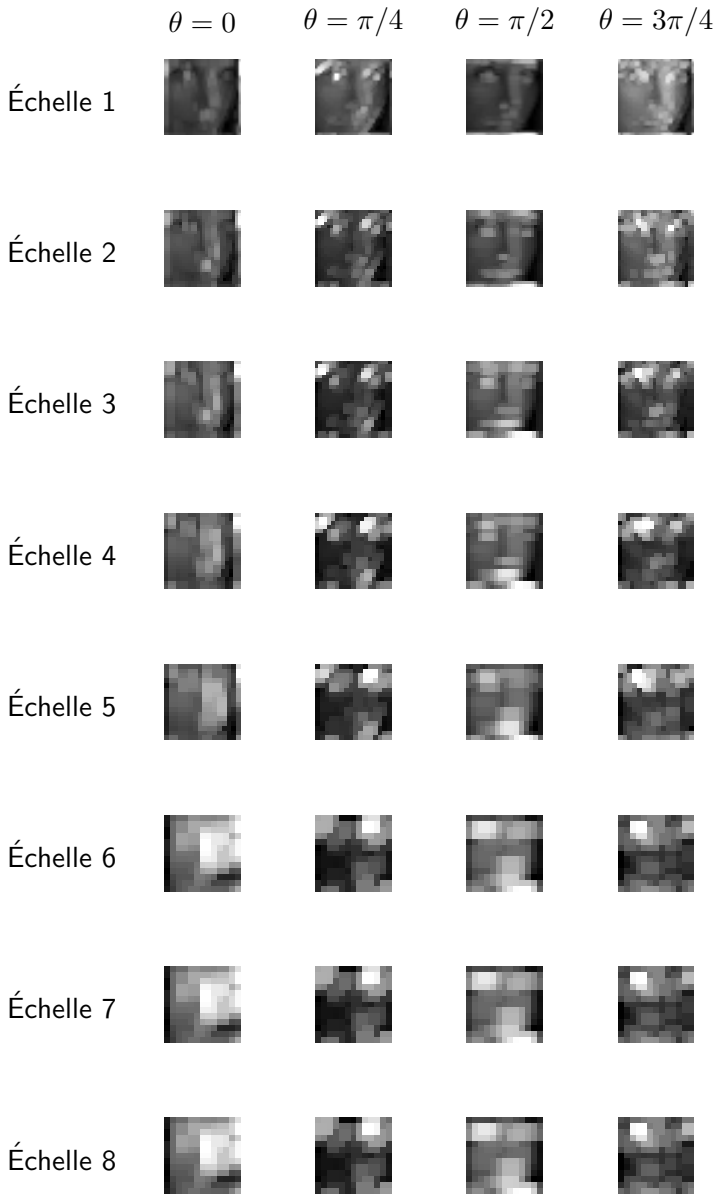


FIGURE B.10: Sorties des C1 pour un visage.

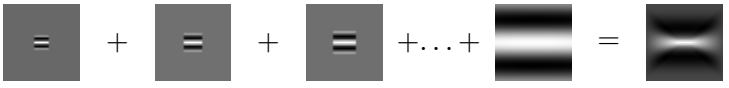


FIGURE B.11: Somme des noyaux de convolutions dans S1.

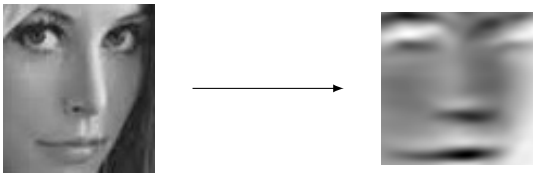


FIGURE B.12: Réponse du filtre unique dans S1 sur un visage.

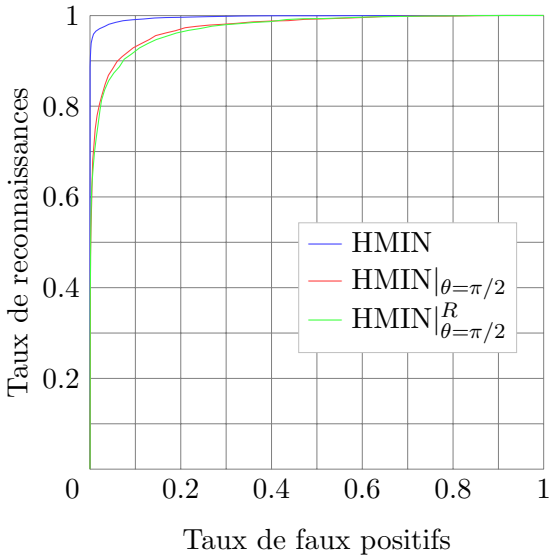


FIGURE B.13: Courbes ROC obtenues avec différentes versions de HMIN sur LFW_Crop.

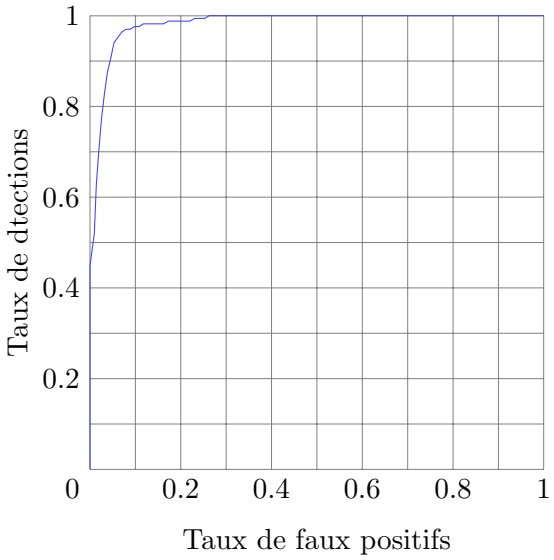


FIGURE B.14: Courbe ROC obtenue avec $\text{HMIN}^{R}_{\theta=\pi/2}$ sur la base CMU.

B.3.2 Détection de piétons

Nous allons commencer par décrire les méthodes avec lesquelles nous allons comparer notre approche. Nous avons choisi de nous comparer avec l'état de l'art du domaine, à savoir le HOG et une implémentation particulière d'un réseau de neurones à convolutions, que nous appellerons ConvNet [145]. La seule différence avec la détection de visages réside dans le fait que, cette fois, nous nous intéressons à des objets verticaux, et donc nous avons choisis de conserver cette fois-ci les filtres d'orientation $\theta = 0$. Nous appellerons les algorithmes ainsi produits $\text{HMIN}_{\theta=0}$ et $\text{HMIN}^{R}_{\theta=0}$.

Méthode	Taux de faux positifs (%)	Complexité (OP)		Empreinte mémoire	Taille d'entrée
		Scanning	Classification		
VJ	5.32×10^{-5} [136]	20.7 M	2.95 k	1.48 MB	24×24
CFF	5×10^{-5} [50]	50.7 M	129.5 k	64.54 MB	36×32
$\text{HMIN}^{R}_{\theta=\pi/2}$	4.5	26.1 M	82.9 k	1.2 MB	32×32

TABLE B.4: Complexité et précision de différentes méthodes de détections de visages. Les taux de faux positifs du CFF et de Viola-Jones ont été lus à partir des courbes ROC de leurs articles respectifs [50, 136], et sont donc approximatifs. Tous les taux de faux positifs correspondent à des taux de détections de 90%. La colonne *Classification* donne la complexité pour la classification d'une image dont la taille est donnée par la colonne *Taille d'entrée*. La colonne *Scanning* donne la complexité de l'algorithme lors d'un scan d'une image VGA complète de dimensions 640×480. Les complexités et empreintes mémoires ont été évaluées pour l'extraction de caractéristiques seulement, sans prendre en compte la classification. Il faut également noter qu'aucune pyramide d'images n'est utilisée ici, pour simplifier les calculs – dans le cas où on en utiliserait une, Viola-Jones demanderait bien moins de ressources que le CFF et HMIN grâce à la représentation en image intégrale.

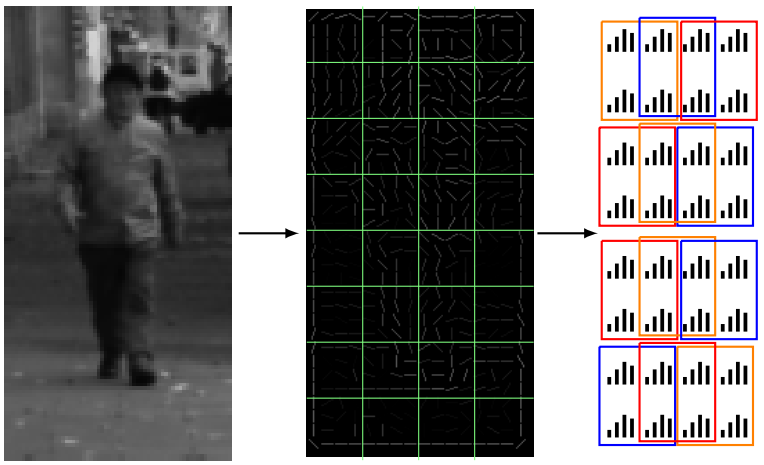


FIGURE B.15: HOG [36].

B.3.2.1 HOG

Cet algorithme consiste en différentes étapes, résumée en Figure B.15. Tout d'abord, des gradients sont calculés à chaque position de l'image d'entrée. Ensuite, des histogrammes locaux sont produits avec ces gradients, afin de produire une carte d'histogrammes. Ces histogrammes sont ensuite normalisés localement, par bloc que 4, afin de produire le vecteurs caractéristique qui sera classifié.

B.3.2.2 ConvNet

Il s'agit simplement d'une implémentation particulière d'un réseaux de neurones à convolutions, avec en prime des étages de normalisation. Cet algorithme opère sur des images représenté en Y'UV. Son architecture est présenté en Figure B.16.

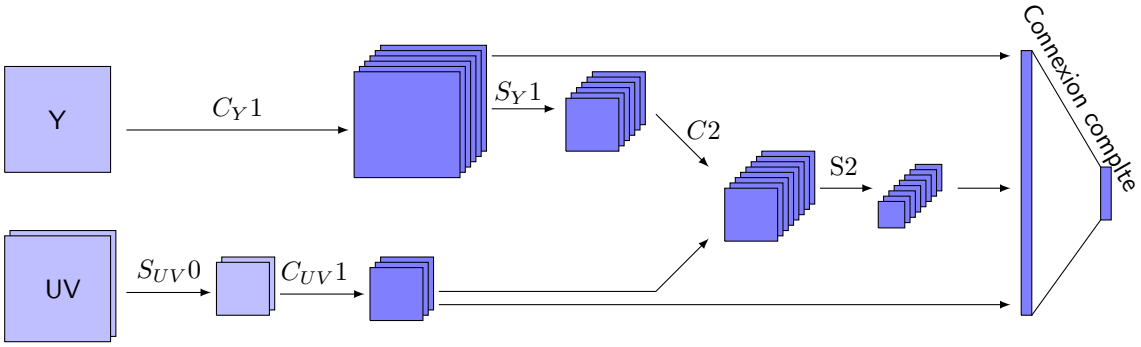


FIGURE B.16: ConvNet pour la détection de piétons [145]. Les couches C_{XX} désignent des couches de convolutions, et les couches S_{XX} désignent des couches de sous-échantillonage.

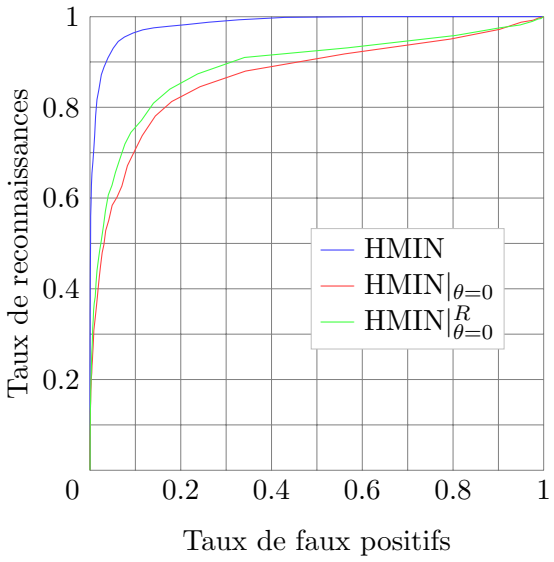


FIGURE B.17: Courbes ROC obtenues avec les descripteurs HMIN sur la base INRIA.

B.3.2.3 Expérimentations

Afin de tester nos algorithmes, nous avons évalué sa précision sur une tâche de détection de piétons sur la base INRIA [36]. Les résultats sont présentés en Figure B.17 et en Table B.5.

B.3.3 Conclusion

Dans cette Section, nous avons présenté notre contribution à l'optimisation d'une méthode d'extraction de caractéristiques. L'algorithme initial est basé sur HMAX, mais n'utilise que ses deux premières couches, S1 et C1. La couche S1 est constituée de 64 filtres

Méthode	Taux de faux positifs (%)	Complexité (OP)		Empreinte mémoire	Taille d'entrée
		Scanning	Classification		
HOG	0.02 [36]	12.96 M	344.7 k	4.37MB	64×128
ConvNet	Voir légende	484.84 M	11 G	63.26MB	78×126
$\text{HMIN}^R_{\theta=0}$	30%	13.05 M	41.45 k	1.2 MB	32×16

TABLE B.5: Complexité et précisions de différentes méthode de détections de personnes. Le taux de faux positifs du HOG a été obtenu à partir des courbes DET présentés dans l'article original [36], et est donc approximatif. Les taux de faux positifs présentés ici correspondent à des taux de détections de 90%. Les résultats concernant le ConvNet ne sont pas directement indiqués ici, en raison du fait que la méthode d'évaluation de sa précision employée dans la littérature est différente de ce qui a été réalisé pour le HOG [145]. Cependant, les contributeurs ont évalué la précision du HOG selon le même critère, et il en ressort que le HOG produit trois fois plus de faux positifs sur cette même base que ConvNet. En raison de ces différences de méthodologies, il est délicat de comparer directement nos résultats avec ceux de la littérature – en revanche, les résultats présentés ici suggèrent un clair désavantage à l'utilisation de $\text{HMIN}^R_{\theta=0}$ pour cette tâche.

de Gabor, avec 16 échelles et 4 orientations différentes. En étudiant les carte de caractéristiques produites par S1 pour différentes tâches spécifiques, nous avons conclus que nous pouvions ne conserver qu'une seule orientation, et sommer les noyaux des convolutions des 16 filtres restants de façon à n'en n'avoir plus qu'un, orienté horizontalement dans le cas de la détection de visages et verticalement dans le cas de la détection de personnes. Nos résultats montrent que notre système a une complexité acceptable, mais sa précision est moindre. Cependant, l'architecture est extrêmement simple, et peut facilement être implantée sur une cible matérielle. De plus, notre architecture est générique : un changement d'applications consiste simplement à changer les poids du noyau de convolution, alors que les autres architectures présentées requièreraient des changements plus en profondeur de l'architecture matérielle. Enfin, l'empreinte mémoire de notre méthode est très faible, ce qui autorise son implantation sur des systèmes ayant de fortes contraintes.

La prochaine Section est dédiée à une proposition d'implantation matérielle de l'algorithme HMAX complet. La dernière Section sera quant à elle dédiée aux discussions finales et aux conclusions générales de nos travaux.

B.4 Implantation matérielle

Dans ce chapitre, nous allons répondre à la seconde problématique, à savoir comment coder efficacement les données manipulées par HMAX, en vue d'une implantation matérielle. Nous allons tout d'abord présenter les optimisations que nous avons

FIGURE B.18: Effet de la dégradation de précision sur l'image d'entrée.

mises en place, en précisant lesquelles sont issues de nos travaux. Ensuite, les résultats d'implantation seront comparés à la littérature.

B.4.1 Optimisations

B.4.1.1 Données en entrée

HMAX traite typiquement des images en niveaux de gris, dans lesquelles les pixels sont codés sur des entiers non-signés 8 bits. Nous proposons d'utiliser moins de bits pour les pixels en entrée, en utilisant toujours des entiers non-signés. Nous avons évalué la précision d'un système de classification d'image constitué de HMAX et d'un classificateur de type GentleBoost pour chacune des précisions allant de 8 bit jusqu'à 1 bit, sur 4 classes de la célèbre base Caltech101: avions, voitures, visages, feuilles et motos. L'effet de la précision des pixels sur la qualité de l'image est présenté en Figure B.18, et sur la précision de la classification en Figure B.19.

B.4.1.2 Filtres de Gabor

Nous nous intéressons maintenant à l'optimisation du codage des filtres de Gabor dans S1. Nous avons testé un codage naïf sur des flottants, puis des entiers signés 16 bits, puis de 8 bits jusqu'à 2 bits. Nous avons également testé un codage sur 1 bit, ou 0

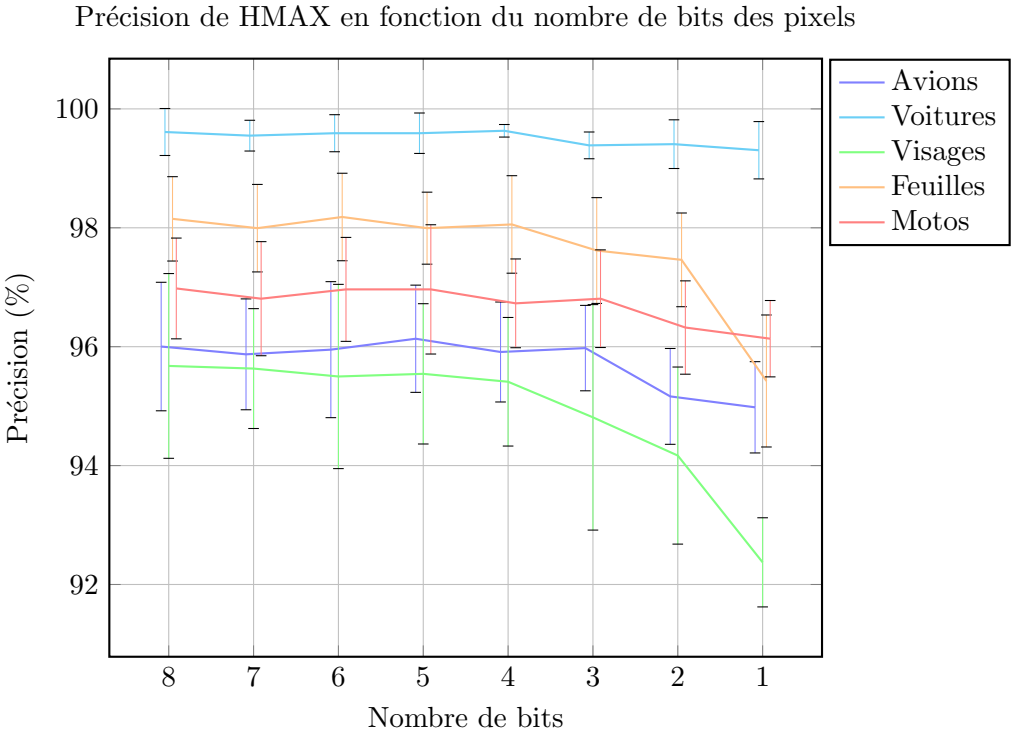


FIGURE B.19: Taux de reconnaissances avec HMAX en fonction de la précision des pixels en entrée.

	Entrée et coefficients des filtres	Méthode de Lloyd	Réduction des patchs de S2	Distance de Manhattan
Avions	95.49 ± 0.81	94.43 ± 0.88	92.07 ± 0.69	91.83 ± 0.63
Voitures	99.45 ± 0.41	99.35 ± 0.40	98.45 ± 0.54	98.16 ± 0.60
Visages	92.97 ± 1.49	90.11 ± 1.05	82.71 ± 1.32	83.35 ± 1.40
Feuilles	96.83 ± 0.79	97.21 ± 0.89	94.61 ± 1.12	93.20 ± 1.42
Motos	95.54 ± 0.79	94.79 ± 0.62	88.83 ± 1.10	89.08 ± 1.31

TABLE B.6: Précision de HMAX en utilisant différentes optimisations.

correspond à -1 et 1 à $+1$. Le nombre de bit pour les pixels de l'image d'entrée est de 2. Cette approche est similaire à ce qui a été proposé dans [167].

B.4.1.3 Autres optimisations

Nous avons appliqué un ensemble d'autres optimisations. La sortie des S1 est compressée sur 2 bits seulement grâce à la méthode de Lloyds, telle que proposée dans [167]. Nous avons également réduit le nombre de vecteurs pré-appris dans S2 grâce à la méthode de Yu *et al.* [46]. De plus, nous avons utilisé une distance de Manhattan au lieu d'une distance Euclidienne dans les opérations de comparaison de motifs de S2. En cumulant ces optimisations avec une précision de 2 bits pour les pixels de l'image d'entrée et de 1 bit pour les filtres de Gabor, nous obtenons les résultats présentés en Table B.6.

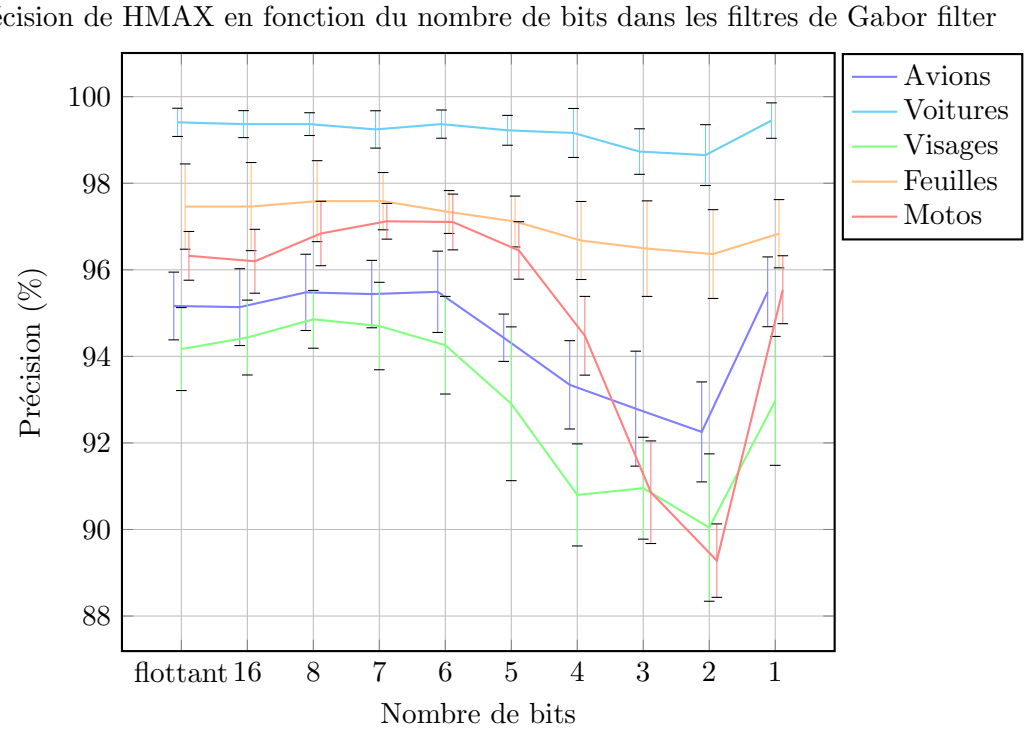


FIGURE B.20: Précisions en fonction du nombres de bits dans les filtres de Gabor de S1, avec 2 bits pour l'image d'entrée.

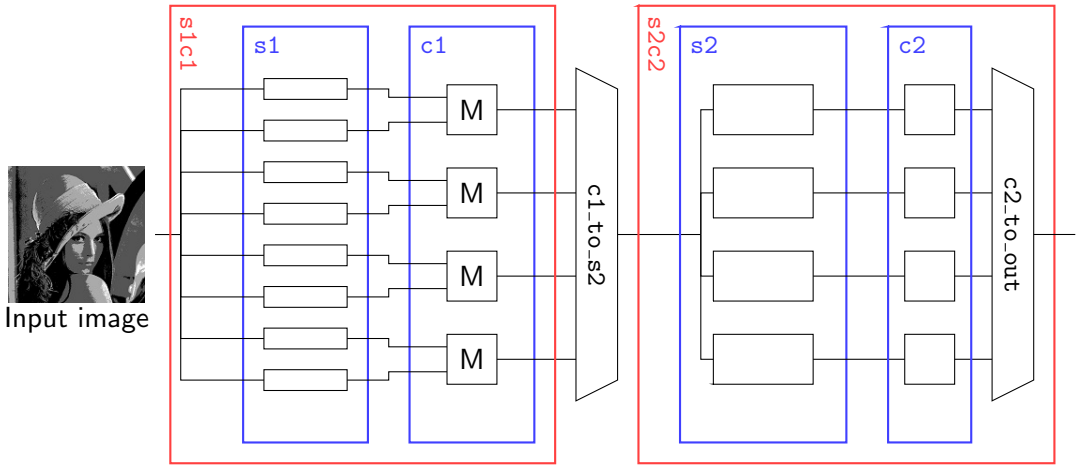


FIGURE B.21: Aperçu du module VHDL HMAX.

B.4.2 Résultats d'implantation

Nous avons utilisé les optimisations présentées ci-dessus pour implanter notre propre version matérielle de HMAX, en VHDL, en ciblant un FPGA Xilinx Artix7-200T. En dehors de ces optimisations, nous avons fait en sorte que notre implantation soit aussi naïve que possible, et nous l'avons comparée avec celle de Orchard *et al.* [99]. L'implantation ne sera pas détaillée ici, mais un schéma général est montré en Figure B.21.

Ressource	Estimation	Disponible	Utilisation (%)
Look-up tables	58204	133800	43.50
Flip-flops	158161	267600	59.10
Inputs/outputs	33	285	11.58
Global buffers	6	32	18.75
Block RAM	254	365	69.59

TABLE B.7: Utilisation des ressources matérielles de HMAX sur un Artix7-200T.

La Table B.7 présente une estimation de l'utilisation des ressources matérielles. Concernant le timing, une étude théorique indique que, sur la base d'une fréquence de l'horloge système à 100 MHz, notre système peut traiter 22.69 images par seconde, contre 193 pour l'implantation présentée en [99]. Cela est dû à une organisation des ressources très différentes, notamment au niveau du multiplexage. Cependant, notre implantation requiert moins de ressources matérielles, et il est important de signaler que nos optimisations et celles proposées par Orchard *et al.* [99] sont parfaitement compatibles.

B.4.3 Conclusion

Dans cette Section, nous avons présenté une série d'optimisations pour HMAX visant à faciliter son implantation matérielle. Notre contribution consiste à diminuer la précision des pixels de l'image d'entrée, diminuer la précision des coefficients des filtres de Gabor et utiliser une distance de Manhattan dans la couche S2 lors des opérations de comparaisons de motifs. Nous utilisons également des méthodes proposées dans la littérature consistant à utiliser l'algorithme de Lloyd pour compresser la sortie de S1, et pour diminuer la complexité de S2. Nous avons montré que ces simplifications n'ont que peu d'impact sur la précision du modèle.

Nous avons ensuite présenté les résultats de l'implantation matérielle, que nous avons voulu aussi naïve que possible en dehors des optimisations proposées ici, puis nous avons comparé le résultat avec la littérature. Il apparaît que notre implantation traite les images significativement moins rapidement que ce qui est proposé dans la littérature ; cependant notre implantation utilise moins de ressources matérielles et nos optimisations sont parfaitement compatibles avec l'implantation de référence. Les travaux futurs consisteront donc à proposer une implantation tirant parti des avantages des deux méthodes, afin de proposer une implantation la plus réduite et avec la plus grande bande passante possible.

B.5 Conclusion

Dans cette thèse, nous avons proposé une solution à un problème d'optimisation d'un algorithme bio-inspiré pour la classification de motifs visuels, avec pour but de l'implanter sur une architecture matérielle dédiée. Notre but était de proposer une architecture facilement embarquable et suffisamment générique pour répondre à différents problèmes. Notre choix s'est porté sur HMAX, en raison de l'unicité de son architecture et de ses performances acceptable même avec un nombre réduit d'examples à apprendre, contrairement à ConvNet.

Notre première contribution consistait à optimiser HMIN, qui est une version allégée de HMAX, pour deux tâches précises, la détection de visages et la détection de piétons, en se basant sur le fait que seules certaines caractéristiques sont utiles. Les performances que nous avons obtenus, pour chacune des deux tâches, sont significativements inférieures à celles proposées dans la littérature – cependant, nous estimons que notre algorithme à l'avantage d'être plus générique, et nous pensons qu'une implémentation matérielle nécessiterait extrêmement peu de ressources.

Notre seconde contribution est de proposer une série d'optimisations pour l'algorithme HMAX complet, principalement basées sur un codage des données efficace. Nous avons montré qu'HMAX ne perdait pas de précisions de manière significative en réduisant la précision des pixels des images d'entrées à 2 bits, et celle des coefficients des filtres de Gabor à 1 seul bit. Bien que cette implantation, naïve en dehors des optimisations nommées ci-dessus, ne permettent pas de traiter une quantité d'images équivalentes à ce qu'il se fait dans la littérature, nos optimisations sont parfaitement utilisables en conjonctions avec celles de l'algorithme de référence, ce qui produirait une implantation particulièrement compact et rapide de cet algorithme – ce qui sera réalisé dans des recherches futures.

Publications

1. Boisard, O., Sauvage, G., Brousse, O., Paindavoine, M., *Implémentation optimisée dun classifieur neuronal pour la détection en temps réel de personnes à terre.* GRETSI 2015.

2. Paindavoine, M., Boisard, O., Carbon, A., Philippe, J.-M., Brousse, O., *NeuroDSP Accelerator for Face Detection Application*, GLSVLSI 2015.

3. Boisard, O., Paindavoine, M., Brousse, O., Doussot, M., *Optimizations for a bio-inspired algorithm towards implementation on embedded platforms*, NeuCOMP, DATE 2015.

4. Boisard, O., Brousse, O., Paindavoine, M., *Processeur 4 bits.* Patent filed to the National Institute of Industrial Property (France), December 22th 2015, N. 1563089, N/REF-V/REF G502-B-45133 FR.

Bibliography

[1] A. M. TURING. I.computing machinery and intelligence. *Mind*, LIX(236):433–460, 1950. doi: 10.1093/mind/LIX.236.433. URL http://mind.oxfordjournals.org/content/LIX/236/433.short.

[2] Yves Langeron, Michel Doussot, David J Hewson, and Jacques Duchêne. Classifying nir spectra of textile products with kernel methods. *Engineering Applications of Artificial Intelligence*, 20(3):415–427, 2007.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 0028-0836. doi: 10.1038/nature16961. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html#auth-1.

[4] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1701–1708, June 2014. doi: 10.1109/CVPR.2014.220.

[5] Michel Paindavoine, Olivier Boisard, Alexandre Carbon, Jean-Marc Philippe, and Olivier Brousse. NeuroDSP Accelerator for Face Detection Application. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 211–215, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3474-7. doi: 10.1145/2742060.2743769. URL http://doi.acm.org/10.1145/2742060.2743769.

[6] E. Fix and J. L. Hodges. Discriminatory analysis, nonparametric discrimination. *US Air Force School of Aviation Medicine*, Technical Report 4, February 1951.

[7] Geoffrey E. Hinton and Terrence J. Sejnowski. Optimal perceptual inference. In *In CVPR, Washington DC*, 1983.

[8] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, January 1985. ISSN 0364-0213. doi: 10.1016/S0364-0213(85)80012-4. URL http://www.sciencedirect.com/science/article/pii/S0364021385800124.

[9] David E. Rumelhart, James L. McClelland, and P. D. P. Research Group P. D. P. Research Group. *Parallel Distributed Processing - Explorations in the Microstructure of Cognition: Foundations*. MIT Press, new edition edition, January 1986. ISBN 978-0-262-68053-0.

[10] Yoshua Bengio Hugo Larochelle. Classification using discriminative restricted Boltzmann machines. *Proceedings of the 25th International Conference on Machine Learning*, pages 536–543, 2008. doi: 10.1145/1390156.1390224.

[11] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558, April 1982. ISSN 0027-8424. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC346238/.

[12] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, January 1982. ISSN 0340-1200, 1432-0770. doi: 10.1007/BF00337288. URL http://link.springer.com/article/10.1007/BF00337288.

[13] Laurene V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms And Applications: United States Edition*. Pearson, Englewood Cliffs, NJ, dition: 1 edition, 1993. ISBN 978-0-13-334186-7.

[14] Frank Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Spartan Books, 1962.

[15] Michael A. Arbib. *Brains, Machines, and Mathematics*. Springer-Verlag New York Inc., New York, NY, 2nd ed. 1987. softcover reprint of the original 2nd ed. 1987 edition, October 2011. ISBN 978-1-4612-9153-4.

[16] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley Publishing Company, 1991. ISBN 978-0-201-51560-2.

[17] Marvin Minsky and Seymour A. Papert. *Perceptrons - An Intro to Computational Geometry Exp Ed*. MIT Press, Cambridge, Mass, revised edition edition, January 1988. ISBN 978-0-262-63111-2.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Internal Representations by Error Propagation. Technical report, University of California and Carnegie-Mellon University, September 1985.

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986. doi: 10.1038/323533a0. URL http://www.nature.com/nature/journal/v323/n6088/abs/323533a0.html.

[20] D. S Broomhead, D Lowe, and Royal Signals and Radar Establishment (Great Britain). *Radial basis functions, multi-variable functional interpolation and adaptive networks*. Royals Signals & Radar Establishment, Malvern, Worcs., 1988.

[21] D.S. Broomhead and D. Lowe. Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems*, 2:321–355, 1988.

[22] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. ISSN 0022-3751. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/.

[23] Timothe Masquelier and Simon J Thorpe. Unsupervised Learning of Visual Features through Spike Timing Dependent Plasticity. *PLoS Comput Biol*, 3(2):e31, 2007. doi: 10.1371/journal.pcbi.0030031. URL http://dx.plos.org/10.1371/journal.pcbi.0030031.

[24] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. ISBN 0-387-31073-8.

[25] David Opitz and Richard Maclin. Popular ensemble methods: an empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[26] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006. ISSN 1531-636X. doi: 10.1109/MCAS.2006.1688199.

[27] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, November 2009. ISSN 0269-2821, 1573-7462. doi: 10.1007/s10462-009-9124-7. URL http://link.springer.com/article/10.1007/s10462-009-9124-7.

[28] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, June 1990. ISSN 0885-6125, 1573-0565. doi: 10.1007/BF00116037. URL http://link.springer.com/article/10.1007/BF00116037.

[29] Leo Breiman. Arcing classifier (with discussion and a rejoinder by the author). *The Annals of Statistics*, 26(3):801–849, June 1998. ISSN 0090-5364, 2168-8966. doi: 10.1214/aos/1024691079. URL http://projecteuclid.org/euclid.aos/1024691079.

[30] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001. CVPR 2001*, volume 1, pages I–511–I–518 vol.1, 2001. doi: 10.1109/CVPR.2001.990517.

[31] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(3):411–426, 2007. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4069258.

[32] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94.

[33] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008. ISSN 1077-3142. doi: 10.1016/j.cviu.2007.09.014. URL http://www.sciencedirect.com/science/article/pii/S1077314207001555.

[34] Edward Rosten and Tom Drummond. Machine Learning for High-Speed Corner Detection. In *Lecture notes in computer science*, pages 430–443. Springer, 2006. ISBN 978-3-540-33832-1. URL http://cat.inist.fr/?aModele=afficheN&cpsidt=20046132.

[35] Adam Schmidt, Marek Kraft, and Andrzej Kasiski. An Evaluation of Image Feature Detectors and Descriptors for Robot Navigation. In Leonard Bolc, Ryszard Tadeusiewicz, Leszek J. Chmielewski, and Konrad Wojciechowski, editors, *Computer Vision and Graphics*, number 6375 in Lecture Notes in Computer Science, pages 251–259. Springer Berlin Heidelberg, September 2010. ISBN 978-3-642-15906-0 978-3-642-15907-7. URL http://link.springer.com/chapter/10.1007/978-3-642-15907-7_31. DOI: 10.1007/978-3-642-15907-7_31.

[36] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *In CVPR*, pages 886–893, 2005.

[37] Stéphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1998.

[38] J. Bruna and S. Mallat. Invariant Scattering Convolution Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1872–1886, 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2012.230.

[39] L. Sifre and S. Mallat. Rotation, Scaling and Deformation Invariant Scattering for Texture Discrimination. In *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1233–1240, June 2013. doi: 10.1109/CVPR.2013.163.

[40] Edouard Oyallon and Stephane Mallat. Deep Roto-Translation Scattering for Object Classification. In *2015, Conference on Computer Vision and Pattern Recognition, CVPR*, pages 2865–2873, 2015. URL http://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Oyallon_Deep_Roto-Translation_Scattering_2015_CVPR_paper.html.

[41] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nature neuroscience*, 2(11):1019–1025, 1999. URL http://www.nature.com/neuro/journal/v2/n11/abs/nn1199_1019.html.

[42] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, 160:106–154, 1962.

[43] Sharat Chikkerur, Thomas Serre, Cheston Tan, and Tomaso Poggio. What and where: A Bayesian inference theory of attention. *Vision Research*, 50(22):2233–2247, October 2010. ISSN 0042-6989. doi: 10.1016/j.visres.2010.05.013. URL http://www.sciencedirect.com/science/article/pii/S0042698910002348.

[44] Jim Mutch and David G. Lowe. Multiclass Object Recognition with Sparse, Localized Features. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, pages 11–18, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2597-0. doi: 10.1109/CVPR.2006.200. URL http://dx.doi.org/10.1109/CVPR.2006.200.

[45] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision*, 80(1):45–57, 2008. URL http://link.springer.com/article/10.1007/s11263-007-0118-0.

[46] Guoshen Yu and J.-J. Slotine. FastWavelet-Based Visual Classification. In *19th International Conference on Pattern Recognition, 2008. ICPR 2008*, pages 1–5, 2008. doi: 10.1109/ICPR.2008.4761069.

[47] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in Neural Information Processing Systems*, pages 396–404. Morgan Kaufmann, 1990.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

[49] Yann LeCun, Koray Kavukcuoglu, and Clment Farabet. Convolutional networks and applications in vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 253–256. IEEE, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5537907.

[50] Christophe Garcia and Manolis Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(11):1408–1423, 2004. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1335446.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[52] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[53] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[54] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

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

[56] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. Torch7: A Matlab-like Environment for Machine Learning. *hgpu.org*, December 2011. URL http://hgpu.org/?p=6776.

[57] Cliff Woolley Sharan Chetlur. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, 2014.

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

[59] Andrew P. Davison, Daniel Brderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: A Common Interface for Neuronal Network Simulators. *Frontiers in Neuroinformatics*, 2, January 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2634533/.

[60] HansEkkehard Plesser, Markus Diesmann, Marc-Oliver Gewaltig, and Abigail Morrison. Nest: the neural simulation tool. In Dieter Jaeger and Ranu Jung, editors, *Encyclopedia of Computational Neuroscience*, pages 1849–1852. Springer New York, 2015. ISBN 978-1-4614-6674-1. doi: 10.1007/978-1-4614-6675-8_258. URL http://dx.doi.org/10.1007/978-1-4614-6675-8_258.

[61] Dejan Pecevski, Thomas Natschlger, Klaus Schuch, Dejan Pecevski, Thomas Natschlger, and Klaus Schuch. PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Frontiers in Neuroinformatics*, 3:11, 2009. doi: 10.3389/neuro.11.011.2009. URL http://journal.frontiersin.org/article/10.3389/neuro.11.011.2009/abstract.

[62] Dan Goodman and Romain Brette. Brian: A Simulator for Spiking Neural Networks in Python. *Frontiers in Neuroinformatics*, 2, November 2008. ISSN 1662-5196. doi: 10.3389/neuro.11.005.2008. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2605403/.

[63] Dan F. M. Goodman and Romain Brette. The Brian Simulator. *Frontiers in Neuroscience*, 3(2):192–197, September 2009. ISSN 1662-4548. doi: 10.3389/neuro.01.026.2009. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2751620/.

[64] Jim Mutch, Ulf Knoblich, and Tomaso Poggio. CNS: a GPU-based framework for simulating cortically-organized networks. Technical Report MIT-CSAIL-TR-2010-013 / CBCL-286, Massachusetts Institute of Technology, Cambridge, MA, February 2010.

[65] Helmut Sedding, Ferdinand Deger, Holger Dammertz, Jan Bouecke, and Hendrik Lensch. Massively Parallel Multiclass Object Recognition. In *Proceedings of the VMV 2010*, pages 251–257, 2010.

[66] R. Uetz and S. Behnke. Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009. ICIS 2009*, volume 1, pages 536–541, November 2009. doi: 10.1109/ICICISYS.2009.5357786.

[67] Christopher J.C. Burges, Yann LeCun, and Corinna Cortes. Mnist database. http://yann.lecun.com/exdb/mnist/, Accessed: 2016-04-11.

[68] Yann LeCun, Huang Fu Jie, and Léon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. *CVPR*, 2004.

[69] Junchul Kim, Eunsoo Park, Xuenan Cui, Hakil Kim, and W.A. Gruver. A fast feature extraction in object recognition using parallel processing on CPU and GPU. In *IEEE International Conference on Systems, Man and Cybernetics, 2009. SMC 2009*, pages 3842–3847, October 2009. doi: 10.1109/ICSMC.2009.5346612.

[70] Sbastien Courroux, Stphane Chevobbe, Mehdi Darouich, and Michel Paindavoine. Use of wavelet for image processing in smart cameras with low hardware resources. *Journal of Systems Architecture*, 59(10):826–832, November 2013. ISSN 13837621. doi: 10.1016/j.sysarc.2013.07.007. URL http://linkinghub.elsevier.com/retrieve/pii/S1383762113001318.

[71] F. Galluppi, C. Denk, M.C. Meiner, T.C. Stewart, L.A. Plana, C. Eliasmith, S. Furber, and J. Conradt. Event-based neural computing on an autonomous mobile platform. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2862–2867, 2014. doi: 10.1109/ICRA.2014.6907270.

[72] Olivier Brousse, Michel Paindavoine, and Christian Gamrat. Neuro-inspired learning of low-level image processing tasks for implementation based on nano-devices. In *International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, 2010. doi: 10.1109/DTIS.2010.5487553.

[73] D. Chabi, Weisheng Zhao, D. Querlioz, and J.-O. Klein. Robust neural logic block (NLB) based on memristor crossbar array. In *2011 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 137–143, June 2011. doi: 10.1109/NANOARCH.2011.5941495.

[74] Hyejung Choi, Heesoo Jung, Joonmyoung Lee, Jaesik Yoon, Jubong Park, Dongjun Seong, Wootae Lee, Musarrat Hasan, Gun-Young Jung, and Hyunsang Hwang. An electrically modifiable synapse array of resistive switching memory. *Nanotechnology*, 20(34):345201, 2009. ISSN 0957-4484. doi: 10.1088/0957-4484/20/34/345201. URL http://stacks.iop.org/0957-4484/20/i=34/a=345201.

[75] M. He, J.O. Klein, and E. Belhaire. Design and electrical simulation of on-chip neural learning based on nanocomponents. *Electronics Letters*, 44(9):575–575, April 2008. ISSN 0013-5194. doi: 10.1049/el:20080442.

[76] Si-Yu Liao, J.-M. Retrouvey, G. Agnus, Weisheng Zhao, C. Maneux, S. Fregonese, T. Zimmer, D. Chabi, A. Filoramo, V. Derycke, C. Gamrat, and J.-O. Klein. Design and Modeling of a Neuro-Inspired Learning Circuit Using Nanotube-Based Memory Devices. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(9):2172–2181, September 2011. ISSN 1549-8328. doi: 10.1109/TCSI.2011.2112590.

[77] J.-M. Retrouvey, J.-O. Klein, Si-Yu Liao, and C. Maneux. Electrical simulation of learning stage in OG-CNTFET based neural crossbar. In *2010 5th International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, March 2010. doi: 10.1109/DTIS.2010.5487555.

[78] J.-M. Retrouvey, J.-O. Klein, Si-Yu Liao, and C. Maneux. Electrical simulation of learning stage in OG-CNTFET based neural crossbar. In *2010 5th International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, March 2010. doi: 10.1109/DTIS.2010.5487555.

[79] G. Snider, R. Amerson, D. Carter, H. Abdalla, M.S. Qureshi, J. Leveille, M. Versace, H. Ames, S. Patrick, B. Chandler, A. Gorchetchnikov, and E. Mingolla. From Synapses to Circuitry: Using Memristive Memory to Explore the Electronic Brain. *Computer*, 44(2):21–28, 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.48.

[80] M. Versace and B. Chandler. The brain of a new machine. *IEEE Spectrum*, 47 (12):30–37, December 2010. ISSN 0018-9235. doi: 10.1109/MSPEC.2010.5644776.

[81] Molecular-junction-nanowire-crossbar-based neural network, Accessed: 2016-02-14. URL http://www.google.com/patents/US7359888. Classification aux

tats-Unis 706/26, 706/27, 257/E27.062, 706/15, 977/938; Classification internationale G06N3/00, G06E3/00, H01L27/092, G06F15/18, G06E1/00, G11C13/02, G06N3/063, G11C11/54, G06G7/00; Classification cooprative G11C13/02, G11C13/0014, G06N3/002, G11C11/54, G11C2213/81, G06N3/063, H01L27/092, G11C2213/77, Y10S977/938, B82Y10/00; Classification europenne B82Y10/00, G11C13/00R5C, H01L27/092, G06N3/00B, G11C13/02, G06N3/063, G11C11/54.

[82] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(13):239–255, 2010. ISSN 0925-2312. doi: 10.1016/j.neucom.2010.03.021. URL http://www.sciencedirect.com/science/article/pii/S092523121000216X.

[83] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007. ISSN 1045-9227. doi: 10.1109/TNN.2007.891626.

[84] D. Le Ly and P. Chow. High-Performance Reconfigurable Hardware Architecture for Restricted Boltzmann Machines. *IEEE Transactions on Neural Networks*, 21(11):1780–1792, November 2010. ISSN 1045-9227. doi: 10.1109/TNN.2010.2073481.

[85] Philippe Coussy, Cyrille Chavet, Laura Conde Canencia, and Hugues Nono Wouafo. Fully-Binary Neural Network Model and Optimized Hardware Architectures for Associative Memories. *ACM Journal on Emerging Technologies in Computing Systems*, pages xx–yy, September 2014. URL https://hal.archives-ouvertes.fr/hal-01009473.

[86] Andres Upegui, Yann Thoma, Eduardo Sanchez, Andres Perez-Uribe, Juan Manuel Moreno, and Jordi Madrenas. The perplexus bio-inspired reconfigurable circuit. In *AHS*, pages 600–605, 2007.

[87] Héctor Fabio Restrepo, Ralph Hoffmann, Andres Perez-Uribe, Christof Teuscher, and Eduardo Sanchez. A networked fpga-based hardware implementation of a neural network application. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 337–338. IEEE, 2000.

[88] Fan Yang and M. Paindavoine. Implementation of an rbf neural network on embedded systems: real-time face tracking and identity verification. *IEEE Transactions on Neural Networks*, 14(5):1162–1175, September 2003. ISSN 1045-9227. doi: 10.1109/TNN.2003.816035. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1243718.

[89] F. Benrekia, M. Attari, A. Bermak, and K. Belhout. FPGA implementation of a neural network classifier for gas sensor array applications. In *6th International Multi-Conference on Systems, Signals and Devices, 2009. SSD '09*, pages 1–6, March 2009. doi: 10.1109/SSD.2009.4956804.

[90] T.N.T. Nguyen, K.C. Chandan, B.A.G. Ahmad, and K.S. Yap. FPGA implementation of neural network classifier for partial discharge time resolved data from magnetic probe. In *2011 International Conference on Advanced Power System Automation and Protection (APAP)*, volume 1, pages 451–455, October 2011. doi: 10.1109/APAP.2011.6180444.

[91] Sungho Park, Ahmed Al Maashri, Kevin M. Irick, Aarti Chandrashekhar, Matthew Cotter, Nandhini Chandramoorthy, Michael Debole, and Vijaykrishnan Narayanan. System-On-Chip for Biologically Inspired Vision Applications. *IPSJ Transactions on System LSI Design Methodology*, 5:71–95, 2012. doi: 10.2197/ipsjtsldm.5.71.

[92] A. Al Maashri, M. DeBole, C.-L. Yu, V. Narayanan, and C. Chakrabarti. A hardware architecture for accelerating neuromorphic vision algorithms. In *2011 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 355–360, October 2011. doi: 10.1109/SiPS.2011.6089002.

[93] M. DeBole, Yang Xiao, Chi-Li Yu, A.A. Maashri, M. Cotter, C. Chakrabarti, and V. Narayanan. FPGA-accelerator system for computing biologically inspired feature extraction models. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 751–755, November 2011. doi: 10.1109/ACSSC.2011.6190106.

[94] A.A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Yang Xiao, V. Narayanan, and C. Chakrabarti. Accelerating neuromorphic vision algorithms for recognition. In *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 579–584, June 2012.

[95] Sungho Park, A Al Maashri, Yang Xiao, K.M. Irick, and V. Narayanan. Saliency-driven dynamic configuration of HMAX for energy-efficient multi-object recognition. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 139–144, 2013. doi: 10.1109/ISVLSI.2013.6654636.

[96] Mi Sun Park, S. Kestur, J. Sabarad, V. Narayanan, and M.J. Irwin. An FPGA-based accelerator for cortical object classification. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 691–696, March 2012. doi: 10.1109/DATE.2012.6176559.

[97] Sungho Park, Y.C.P. Cho, K.M. Irick, and V. Narayanan. A reconfigurable platform for the design and verification of domain-specific accelerators. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 108–113, January 2012. doi: 10.1109/ASPDAC.2012.6164928.

[98] S. Kestur, Mi Sun Park, J. Sabarad, D. Dantara, V. Narayanan, Yang Chen, and D. Khosla. Emulating Mammalian Vision on Reconfigurable Hardware. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 141–148, April 2012. doi: 10.1109/FCCM.2012.33.

[99] Garrick Orchard, Jacob G. Martin, R. Jacob Vogelstein, and Ralph Etienne-Cummings. Fast Neuromimetic Object Recognition Using FPGA Outperforms GPU Implementations. *IEEE Transactions on Neural Networks and Learning Systems*, 24(8):1239–1252, August 2013. ISSN 2162-237X, 2162-2388. doi: 10.1109/TNNLS.2013.2253563. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6502724.

[100] Nicolas Farrugia, Franck Mamalet, Sbastien Roux, Fan Yang, and Michel Paindavoine. Fast and robust face detection on a parallel optimized architecture implemented on FPGA. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(4):597–602, 2009. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4797837.

[101] C. Farabet, C. Poulet, J.Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *International Conference on Field Programmable Logic and Applications, 2009. FPL 2009*, pages 32–37, 2009. doi: 10.1109/FPL.2009.5272559.

[102] C. Farabet, C. Poulet, and Y. LeCun. An FPGA-based stream processor for embedded real-time vision with Convolutional Networks. In *2009 IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 878–885, September 2009. doi: 10.1109/ICCVW.2009.5457611.

[103] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116, June 2011. doi: 10.1109/CVPRW.2011.5981829.

[104] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 257–260, 2010. doi: 10.1109/ISCAS.2010.5537908.

[105] Phi-Hung Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello. NeuFlow: Dataflow vision processing system-on-a-chip. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1044–1047, 2012. doi: 10.1109/MWSCAS.2012.6292202.

[106] V. Gokhale, Jonghoon Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 696–701, June 2014. doi: 10.1109/CVPRW.2014.106.

[107] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A Convolutional Network Accelerator. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 199–204, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3474-7. doi: 10.1145/2742060.2743766. URL http://doi.acm.org/10.1145/2742060.2743766.

[108] S.B. Furber, F. Galluppi, S. Temple, and L.A. Plana. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665, 2014. ISSN 0018-9219. doi: 10.1109/JPROC.2014.2304638.

[109] Henry Markram. The blue brain project. *Nature Reviews. Neuroscience*, 7(2):153–160, February 2006. ISSN 1471-003X. doi: 10.1038/nrn1848.

[110] Kai Kupferschmidt. Virtual rat brain fails to impress its critics. *Science*, 350 (6258):263–264, October 2015. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.350.6258.263. URL http://science.sciencemag.org/content/350/6258/263.

[111] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, August 2014. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.1254642. URL http://science.sciencemag.org/content/345/6197/668.

[112] IBM Research: Brain-inspired Chip, Accessed: 2016-02-14. URL http://www.research.ibm.com/articles/brain-chip.shtml.

[113] Jeffrey L. Krichmar, Philippe Coussy, and Nikil Dutt. Large-scale spiking neural networks using neuromorphic hardware compatible models. *J. Emerg. Technol. Comput. Syst.*, 11(4):36:1–36:18, April 2015. ISSN 1550-4832. doi: 10.1145/2629509. URL http://doi.acm.org/10.1145/2629509.

[114] H.M. Hussain, K. Benkrid, and H. Seker. An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA. In *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 205–212, June 2012. doi: 10.1109/AHS.2012.6268651.

[115] Hongying Meng, K. Appiah, A. Hunter, and P. Dickinson. FPGA implementation of Naive Bayes classifier for visual object recognition. In *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 123–128, June 2011. doi: 10.1109/CVPRW.2011.5981831.

[116] Davide Anguita, Luca Carlino, Alessandro Ghio, and Sandro Ridella. A FPGA Core Generator for Embedded Classification Systems. *Journal of Circuits, Systems, and Computers*, 20(2):263–282, 2011. URL http://dblp.uni-trier.de/db/journals/jcsc/jcsc20.html#AnguitaCGR11.

[117] Minghua Shi, A. Bermak, S. Chandrasekaran, and A. Amira. An Efficient FPGA Implementation of Gaussian Mixture Models-Based Classifier Using Distributed Arithmetic. In *13th IEEE International Conference on Electronics, Circuits and Systems, 2006. ICECS '06*, pages 1276–1279, 2006. doi: 10.1109/ICECS.2006.379695.

[118] V. Bonato, E. Marques, and G.A. Constantinides. A Parallel Hardware Architecture for Scale and Rotation Invariant Feature Detection. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(12):1703–1712, 2008. ISSN 1051-8215. doi: 10.1109/TCSVT.2008.2004936.

[119] Lifan Yao, Hao Feng, Yiqun Zhu, Zhiguo Jiang, Danpei Zhao, and Wenquan Feng. An architecture of optimised SIFT feature detection for an FPGA implementation of an image matcher. In *International Conference on Field-Programmable Technology, 2009. FPT 2009*, pages 30–37, 2009. doi: 10.1109/FPT.2009.5377651.

[120] J. Svab, T. Krajnik, J. Faigl, and L. Preucil. FPGA based Speeded Up Robust Features. In *IEEE International Conference on Technologies for Practical Robot Applications, 2009. TePRA 2009*, pages 35–41, November 2009. doi: 10.1109/TEPRA.2009.5339646.

[121] M. Holler, Simon Tam, H. Castro, and R. Benson. An electrically trainable artificial neural network (ETANN) with 10240 'floating gate' synapses. In , *International Joint Conference on Neural Networks, 1989. IJCNN*, pages 191–196 vol.2, 1989. doi: 10.1109/IJCNN.1989.118698.

[122] N. Mauduit, M. Duranton, J. Gobert, and J.-A. Sirat. Lneuro 1.0: a piece of hardware LEGO for building neural network systems. *IEEE Transactions on Neural Networks*, 3(3):414–422, 1992. ISSN 1045-9227. doi: 10.1109/72.129414.

[123] M. Duranton. L-Neuro 2.3: a VLSI for image processing by neural networks. In , *Proceedings of Fifth International Conference on Microelectronics for Neural Networks, 1996*, pages 157–160, 1996. doi: 10.1109/MNNFS.1996.493786.

[124] Kurosh Madani, Ghislain de Trmiolles, and Pascal Tannhof. ZISC-036 Neuroprocessor Based Image Processing. In Jos Mira and Alberto Prieto, editors, *Bio-Inspired Applications of Connectionism*, number 2085 in Lecture Notes in Computer Science, pages 200–207. Springer Berlin Heidelberg, June 2001. ISBN 978-3-540-42237-2 978-3-540-45723-7. URL http://link.springer.com/chapter/10.1007/3-540-45723-2_24. DOI: 10.1007/3-540-45723-2_24.

[125] Cm1k chip. http://www.cognimem.com/products/chips-and-modules/CM1K-Chip/index.html, Accessed: 2016-02-14.

[126] Human brain project. https://www.humanbrainproject.eu/, Accessed: 2016-02-14.

[127] Martin Enserink. A 1 billion brain reboot. *Science*, 347(6229):1406–1407, March 2015. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.347.6229.1406. URL http://science.sciencemag.org/content/347/6229/1406.

[128] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(10):1615–1630, October 2005. ISSN 0162-8828. doi: 10.1109/TPAMI.2005.188.

[129] Plinio Moreno, Manuel J. Marín-Jiménez, Alexandre Bernardino, José Santos-Victor, and Nicolás Pérez de la Blanca. *A Comparative Study of Local Descriptors for Object Category Recognition: SIFT vs HMAX*, pages 515–522. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-72847-4. doi: 10.1007/978-3-540-72847-4_66. URL http://dx.doi.org/10.1007/978-3-540-72847-4_66.

[130] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the Best Multi-Stage Architecture for Object Recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, September 2009. doi: 10.1109/ICCV.2009.5459469.

[131] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going

Deeper with Convolutions. *CoRR*, 2015. URL http://research.google.com/pubs/pub43022.html.

[132] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL http://arxiv.org/abs/1502.01852.

[133] Lapack - linear algrebra package. http://www.netlib.org/lapack/, Accessed: 2016-02-16.

[134] Blas - basic linear algebra subprogram. http://www.netlib.org/blas/, Accessed: 2016-02-16.

[135] CUDA Implementation of a Biologically Inspired Object Recognition System, Accessed: 2016-07-31. URL http://code.google.com/p/cbcl-model-cuda/.

[136] Paul Viola and Michael J. Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004. URL http://link.springer.com/article/10.1023/B:VISI.0000013087.49260.fb.

[137] Itseez. Open source computer vision library. https://github.com/itseez/opencv, 2015. Accessed: 29-2-2016.

[138] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *arXiv:1411.4038 [cs]*, November 2014. URL http://arxiv.org/abs/1411.4038. arXiv: 1411.4038.

[139] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. URL http://arxiv.org/abs/1411.4038.

[140] Olivier Boisard, Michel Paindavoine, Olivier Brousse, and Michel Doussot. Optimizations for a bio-inspired algorithm towards implementation on embedded platforms. *date*, 2015.

[141] Lin-Lin Huang, Akinobu Shimizu, and Hidefumi Kobatake. Robust face detection using Gabor filter features. *Pattern Recognition Letters*, 26(11):1641–1649, August 2005. ISSN 01678655. doi: 10.1016/j.patrec.2005.01.015. URL http://linkinghub.elsevier.com/retrieve/pii/S0167865505000206.

[142] Li Fei-Fei, R. Fergus, and P. Perona. Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. In *Conference on Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04*, pages 178–178, June 2004. doi: 10.1109/CVPR.2004.109.

[143] Kah-Kay Sung, Tomaso Poggio, A. Henry Rowley, Shumeet Baluja, and Takeo Kanade. Cmu frontal face images test set. http://vasc.ri.cmu.edu/idb/html/face/frontal_images/, Accessed: 2016-04-11.

[144] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments, Accessed: 2016-07-31.

[145] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. Lecun. Pedestrian Detection with Unsupervised Multi-stage Feature Learning. In *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3626–3633, June 2013. doi: 10.1109/CVPR.2013.465.

[146] Koray Kavukcuoglu, Pierre Sermanet, Y lan Boureau, Karol Gregor, Michal Mathieu, and Yann Lecun. Learning convolutional feature hierarchies for visual recognition. In *In NIPS10*, 2010.

[147] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto. Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection. In *2012 IEEE Workshop on Signal Processing Systems*, pages 197–202, October 2012. doi: 10.1109/SiPS.2012.57.

[148] M. Jacobsen, Z. Cai, and N. Vasconcelos. FPGA implementation of HOG based pedestrian detector. In *2015 International SoC Design Conference (ISOCC)*, pages 191–192, November 2015. doi: 10.1109/ISOCC.2015.7401776.

[149] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware Architecture for HOG Feature Extraction. In *Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP '09*, pages 1330–1333, September 2009. doi: 10.1109/IIH-MSP.2009.216.

[150] Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, and Konrad Doll. Fpga-based real-time pedestrian detection on high-resolution images. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Portland, USA*, pages 629 – 635, 2013. doi: 10.1109/CVPRW.2013.95.

[151] Pei-Yung Hsiao, Shih-Yu Lin, and Shih-Shinh Huang. An FPGA based human detection system with embedded platform. *Microelectronic Engineering*, 138:42–46, 2015. ISSN 0167-9317. doi: 10.1016/j.mee.2015.01.018. URL http://www.sciencedirect.com/science/article/pii/S0167931715000271.

[152] K. Negi, K. Dohi, Y. Shibata, and K. Oguri. Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm. In *2011*

International Conference on Field-Programmable Technology (FPT), pages 1–8, 2011. doi: 10.1109/FPT.2011.6132679.

[153] M. Komorkiewicz, M. Kluczewski, and M. Gorgon. Floating point HOG implementation for real-time multiple object detection. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 711–714, 2012. doi: 10.1109/FPL.2012.6339159.

[154] C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods. Histogram of oriented gradients front end processing: An FPGA based processor approach. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, October 2014. doi: 10.1109/SiPS.2014.6986093.

[155] Tam Phuong Cao, Guang Deng, and D. Mulligan. Implementation of real-time pedestrian detection on FPGA. In *2008 23rd International Conference Image and Vision Computing New Zealand*, pages 1–6, November 2008. doi: 10.1109/IVCNZ.2008.4762094.

[156] S. Lee, H. Son, J. C. Choi, and K. Min. HOG feature extractor circuit for real-time human and vehicle detection. In *TENCON 2012 - 2012 IEEE Region 10 Conference*, pages 1–5, November 2012. doi: 10.1109/TENCON.2012.6412287.

[157] P. Y. Chen, C. C. Huang, C. Y. Lien, and Y. H. Tsai. An Efficient Hardware Implementation of HOG Feature Extraction for Human Detection. *IEEE Transactions on Intelligent Transportation Systems*, 15(2):656–662, April 2014. ISSN 1524-9050. doi: 10.1109/TITS.2013.2284666.

[158] F. Karakaya, H. Altun, and M. A. Cavuslu. Implementation of HOG algorithm for real time object recognition applications on FPGA based embedded system. In *2009 IEEE 17th Signal Processing and Communications Applications Conference*, pages 508–511, April 2009. doi: 10.1109/SIU.2009.5136444.

[159] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware Architecture for HOG Feature Extraction. In *Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP '09*, pages 1330–1333, September 2009. doi: 10.1109/IIH-MSP.2009.216.

[160] H. T. Ngo, R. C. Tompkins, J. Foytik, and V. K. Asari. An area efficient modular architecture for real- time detection of multiple faces in video stream. In *2007 6th International Conference on Information, Communications Signal Processing*, pages 1–5, 2007. doi: 10.1109/ICICS.2007.4449885.

[161] C. Cheng and C. S. Bouganis. An FPGA-based object detector with dynamic workload balancing. In *2011 International Conference on Field-Programmable Technology (FPT)*, pages 1–4, 2011. doi: 10.1109/FPT.2011.6132723.

[162] Changjian Gao and Shih-Lien Lu. Novel FPGA based Haar classifier face detection algorithm acceleration. In *2008 International Conference on Field Programmable Logic and Applications*, pages 373–378, September 2008. doi: 10.1109/FPL.2008.4629966.

[163] S. Das, A. Jariwala, and P. Engineer. Modified architecture for real-time face detection using FPGA. In *2012 Nirma University International Conference on Engineering (NUiCONE)*, pages 1–5, 2012. doi: 10.1109/NUICONE.2012.6493235.

[164] Sbastien Roux, Franck Mamalet, and Christophe Garcia. Embedded Convolutional Face Finder. In *IEEE International Conference on Multimedia & Expo (ICME2006)*, pages 285–288, August 2006. ISBN 1-4244-0367-7. doi: 10.1109/ICME.2006.262454. URL http://liris.cnrs.fr/publis/?id=6107.

[165] Sbastien Roux, Franck Mamalet, and Christophe Garcia. Embedded facial image processing with Convolutional Neural Networks. In *IEEE Circuits and Systems Conference (ISCAS 2010)*, pages 261–264, June 2010. URL http://liris.cnrs.fr/publis/?id=6072.

[166] Paul Viola, Michael J Jones, and Daniel Snow. Detecting pedestrians using patterns of motion and appearance. *International Journal of Computer Vision*, 63 (2):153–161, 2005.

[167] Sharat Chikkerur and Tomaso Poggio. Approximations in the HMAX Model, April 2011. URL http://dspace.mit.edu/handle/1721.1/62293.

[168] Thomas Serre, Aude Oliva, and Tomaso Poggio. A feedforward architecture accounts for rapid categorization. *Proceedings of the National Academy of Sciences*, 104(15):6424–6429, April 2007. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.0700622104. URL http://www.pnas.org/content/104/15/6424.

[169] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[170] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *arXiv:1511.00363 [cs]*, November 2015. URL http://arxiv.org/abs/1511.00363. arXiv: 1511.00363.

[171] Hong-Phuc Trinh, Marc Duranton, and Michel Paindavoine. Efficient Data Encoding for Convolutional Neural Network Application. *ACM Trans. Archit. Code Optim.*, 11(4):49:1–49:21, January 2015. ISSN 1544-3566. doi: 10.1145/2685394. URL http://doi.acm.org/10.1145/2685394.

[172] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–136, 1982. doi: 10.1109/TIT.1982.1056489.

[173] G. Roe. Quantizing for minimum distortion (corresp.). *IEEE Trans. Inf. Theor.*, 10(4):384–385, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1964.1053693. URL http://dx.doi.org/10.1109/TIT.1964.1053693.

[174] Mohamad T. Musavi, Wahid Ahmed, Khue Hiang Chan, Kathleen B. Faris, and Donald M. Hummels. On the training of radial basis function classifiers. *Neural networks*, 5(4):595–603, 1992. URL http://www.sciencedirect.com/science/article/pii/S0893608005800383.