HAL
open science

Towards real-time image understanding with convolutional networks

Clément Farabet

To cite this version:

Clément Farabet. Towards real-time image understanding with convolutional networks. Computation and Language [cs.CL]. Université Paris-Est, 2013. English. NNT : 2013PEST1083 . tel-01327221

HAL Id: tel-01327221

https://pastel.hal.science/tel-01327221v1

Submitted on 6 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École doctorale MSTIC
Université Paris-Est

A thesis submitted for the degree of
Docteur de l'université Paris-Est en Informatique

Clément Farabet

Directeur de thèse: Laurent Najman
Co-Directeur de thèse: Yann LeCun

Towards Real-Time Image Understanding with Convolutional Networks

December 19, 2013

Jury:

Yoshua Bengio	(Président & Rapporteur)
Léon Bottou	(Rapporteur)
Eugenio Culurciello	(Examinateur)
Sumit Chopra	(Examinateur)
Laurent Najman	(Directeur de thèse)
Yann LeCun	(Co-Directeur de thèse)

Abstract

One of the open questions of artificial computer vision is how to produce good internal representations of the visual world. What sort of internal representation would allow an artificial vision system to detect and classify objects into categories, independently of pose, scale, illumination, conformation, and clutter? More interestingly, how could an artificial vision system *learn* appropriate internal representations automatically, the way animals and humans seem to learn by simply looking at the world?

Another related question is that of computational tractability, and more precisely that of computational efficiency. Given a good visual representation, how efficiently can it be trained, and used to encode new sensorial data. Efficiency has several dimensions: power requirements, processing speed, and memory usage.

In this thesis I present three new contributions to the field of computer vision: (1) a multiscale deep convolutional network architecture to easily capture long-distance relationships between input variables in image data, (2) a tree-based algorithm to efficiently explore multiple segmentation candidates, to produce maximally confident semantic segmentations of images, (3) a custom dataflow computer architecture optimized for the computation of convolutional networks, and similarly dense image processing models. All three contributions were produced with the common goal of getting us closer to real-time image understanding.

Scene parsing consists in labeling each pixel in an image with the category of the object it belongs to. In the first part of this thesis, I propose a method that uses a multiscale convolutional network trained from raw pixels to extract dense feature vectors that encode regions of multiple sizes centered on each pixel. The method alleviates the need for engineered features. In

parallel to feature extraction, a tree of segments is computed from a graph of pixel dissimilarities. The feature vectors associated with the segments covered by each node in the tree are aggregated and fed to a classifier which produces an estimate of the distribution of object categories contained in the segment. A subset of tree nodes that cover the image are then selected so as to maximize the average "purity" of the class distributions, hence maximizing the overall likelihood that each segment contains a single object. The system yields record accuracies on several public benchmarks.

The computation of convolutional networks, and related models heavily relies on a set of basic operators that are particularly fit for dedicated hardware implementations. In the second part of this thesis I introduce a scalable dataflow hardware architecture optimized for the computation of general-purpose vision algorithms—*neuFlow*—and a dataflow compiler—*luaFlow*—that transforms high-level flow-graph representations of these algorithms into machine code for neuFlow. This system was designed with the goal of providing real-time detection, categorization and localization of objects in complex scenes, while consuming 10 Watts when implemented on a Xilinx Virtex 6 FPGA platform, or about ten times less than a laptop computer, and producing speedups of up to 100 times in real-world applications (results from 2011).

To my wife, Domitille Farabet.

Acknowledgements

I would like to thank Prof. Yann LeCun for welcoming me into his intellectual circle, for sharing his long-term vision with me, and for continuously defining and redefining an exciting research field. While working on my PhD thesis, I enjoyed the freedom of exploration, while always benefiting from his full support and patience. It has been a great privilege and unique experience to grow as a researcher under his guidance.

I would like to thank Prof. Laurent Najman, my thesis adviser, for his continuous guidance during my thesis work. I'm infinitely grateful for his advice, patience and mentoring.

I would like to thank Eugenio Culurciello, Ronan Collobert, Camille Couprie, Marco Scoffier, Koray Kavukcuoglu, and Berin Martini for having been great collaborators, and all the people I have had fruitful discussions with, at Prof. Yann LeCun's lab and outside.

I would also like to thank the members of my jury for their feedback on my thesis.

Finally, I am indebted to my wife, for her patience, and my parents, for putting me on the right tracks.

Context

The research that led to this PhD thesis was conducted at the Courant Institute of Mathematical Sciences, New York University, over the course of 5 years, between 2008 and 2013. I officially registered as a PhD student at Université Paris-Est from 2010 to 2013. This research was done in close collaboration with Prof. Yann LeCun at NYU, and with Prof. Laurent Najman at Université Paris-Est. The work on neuFlow (Chapter 3) started as a Master's thesis/project in 2008.

Publications

Journals

C. Farabet, C. Couprie, L. Najman and Y. LeCun, "Learning Hierarchical Features for Scene Labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, in press, 2013.

C. Farabet, R. Paz, J. Perez-Carrasco, C. Zamarreno, A. Linares-Barranco, Y. LeCun, E. Culurciello, T. Serrano-Gotarredona and B. Linares-Barranco, "Comparison Between Frame-Constrained Fix-Pixel-Value and Frame-Free Spiking-Dynamic-Pixel ConvNets for Visual Processing", in *Frontiers in Neuroscience*, 2012.

C. Farabet, Y. LeCun, K. Kavukcuoglu, B. Martini, P. Akselrod, S. Talay, and E. Culurciello, "Large-Scale FPGA-Based Convolutional Networks", in R. Bekkerman, M. Bilenko, and J. Langford (Ed.), *Scaling Up Machine Learning*, Cambridge University Press, 2011.

International Conferences

C. Couprie, C. Farabet, L. Najman, Y. LeCun, "Indoor Semantic Segmentation using depth information", in *Proceedings of the International Conference on Learning Representations*, May 2013.

C. Culurciello, J. Bates, A. Dundar, J. Carrasco, C. Farabet, "Clustering Learning for Robotic Vision", ArXiv preprint, January 2013, in *Proceedings of the International Conference on Learning Representations*, May 2013.

C. Farabet, C. Couprie, L. Najman, Y. LeCun, "Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers", in *Proc. of*

the International Conference on Machine Learning (ICML'12), Edinburgh, Scotland, 2012. Video: `http://techtalks.tv/talks/57300/`

Phi-Hung Pham, Darko Jelaca, Clement Farabet, Berin Martini, Yann LeCun and Eugenio Culurciello, "NeuFlow: Dataflow Vision Processing System-on-a-Chip", in *IEEE International Midwest Symposium on Circuits and systems*, IEEE MWSCAS, 2012, Boise, Idaho, USA.

C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision", *in Proc. of the Fifth IEEE Workshop on Embedded Computer Vision (ECV'11 @ CVPR'11)*, IEEE, Colorado Springs, 2011. Invited Paper.

C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun and E. Culurciello, "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems", in *International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, Paris, 2010.

Y. LeCun, K. Kavukcuoglu and C. Farabet, "Convolutional Networks and Applications in Vision", in *International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, Paris, 2010.

C. Farabet, C. Poulet and Y. LeCun, "An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks", in *Proc. of the Fifth IEEE Workshop on Embedded Computer Vision (ECV'09 @ ICCV'09)*, IEEE, Kyoto, 2009.

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 (FPL'09)*, IEEE, Prague, 2009.

Software, Patent

R. Collobert, K. Kavukcuoglu, C. Farabet, "Torch7: A Matlab-like Environment for Machines Learning", in *Big Learning Workshop (@ NIPS'11)*, Sierra Nevada, Spain, 2011. `http://www.torch.ch`

C. Farabet, Y. LeCun, "Runtime Reconfigurable Dataflow Processor", filed on May 24, 2012. US Patent Application Number 13/479,742.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Representation Learning with Deep Networks 3

1.1.1 Deep Network Architectures 3

1.1.1.1 Multilayer Perceptrons 4

1.1.1.2 Convolutional Networks 5

1.1.1.3 Encoders + Decoders = Auto-encoders 9

1.1.2 Learning: Parameter Estimation 11

1.1.2.1 Loss Function, Objective 11

1.1.2.2 Optimization 12

1.2 Hierarchical Segmentations, Structured Prediction 12

1.2.1 Hierarchical Segmentations 13

1.2.1.1 Graph Representation 13

1.2.1.2 Minimum Spanning Trees 15

1.2.1.3 Dendograms 16

1.2.1.4 Segmentations 16

1.2.2 Structured Prediction 17

1.2.2.1 Graphical Models 18

1.2.2.2 Learning: Parameter Estimation 19

1.3 Dataflow Computing 20

2 Image Understanding: Scene Parsing **25**
2.1 Introduction 25
2.2 A Model for Scene Understanding 25
2.2.1 Introduction 25
2.2.2 Multiscale feature extraction for scene parsing 29
2.2.2.1 Scale-invariant, scene-level feature extraction 30
2.2.2.2 Learning discriminative scale-invariant features 32
2.2.3 Scene labeling strategies 33
2.2.3.1 Superpixels 33
2.2.3.2 Conditional Random Fields 34
2.2.3.3 Parameter-free multilevel parsing 36
2.2.4 Experiments 41
2.2.4.1 Multiscale feature extraction 44
2.2.4.2 Parsing with superpixels 45
2.2.4.3 Multilevel parsing 46
2.2.4.4 Conditional random field 47
2.2.4.5 Some comments on the learned features 47
2.2.4.6 Some comments on real-world generalization 47
2.2.5 Discussion and Conclusions 48

3 A Hardware Platform for Real-time Image Understanding **57**
3.1 Introduction 57
3.2 Learning Internal Representations 58
3.2.1 Convolutional Networks 59
3.2.2 Unsupervised Learning of ConvNets 62
3.2.2.1 Unsupervised Training with Predictive Sparse Decomposition 62
3.2.2.2 Results on Object Recognition 63
3.2.2.3 Connection with Other Approaches in Object Recognition 64
3.3 A Dedicated Digital Hardware Architecture 65
3.3.1 A Data-Flow Approach 66
3.3.1.1 On Runtime Reconfiguration 68
3.3.2 An FPGA-Based ConvNet Processor 72

3.3.2.1 Specialized Processing Tiles 74
3.3.2.2 Smart DMA Implementation 75
3.3.3 Compiling ConvNets for the ConvNet Processor 76
3.3.4 Application to Scene Understanding 78
3.3.5 Performance 81
3.3.6 Precision 84

4 Discussion **87**

References **89**

List of Figures

1.1 Common ConvNet Blocks 7

1.2 Graphs with local connectivity 14

1.3 Gradient graph 14

1.4 Minimum Spanning Tree of a Graph 15

1.5 Dendogram of an MST 16

1.6 Cutting the Dendogram = Segmenting 17

1.7 Von Neumann Architecture 22

1.8 Our Proposed Dataflow Architecture 24

2.1 Model overview 27

2.2 Superpixels 34

2.3 CRFs 35

2.4 Optimal Cover 37

2.5 Optimal Cover on a Tree: 39

2.6 Max Sampling 40

2.7 Scene Parsing Results 52

2.8 More Scene Parsing Results 53

2.9 Scene Parsing Results with More Classes 53

2.10 Learned Filters 54

2.11 Real-time scene parsing in natural conditions 55

3.1 Architecture of a typical convolutional network for object recognition. This implements a convolutional feature extractor and a linear classifier for generic N-class object recognition. Once trained, the network can be computed on arbitrary large input images, producing a classification map as output. 59
3.2 A data-flow computer. A set of runtime configurable processing tiles are connected on a 2D grid. They can exchange data with their 4 neighbors and with an off-chip memory via global lines. 67
3.3 The grid is configured for a complex computation that involves several tiles: the 3 top tiles perform a 3×3 convolution, the 3 intermediate tiles another 3×3 convolution, the bottom left tile sums these two convolutions, and the bottom centre tile applies a function to the result. 71
3.4 Overview of the ConvNet Processor system. A grid of multiple full-custom Processing Tiles tailored to ConvNet operations, and a fast streaming memory interface (Smart DMA). 73
3.5 Scene Parsing on FPGAs 79
3.6 Compute time for a typical ConvNet (as seen in Figure 3.1). 82
3.7 Quantization effect on trained networks: the x axis shows the fixed point position, the y axis the percentage of weights being zeroed after quantization. 86

List of Tables

2.1 Performance of our system on the Stanford Background dataset 42

2.2 Performance of our system on the SIFT Flow dataset 42

2.3 Performance of our system on the Barcelona dataset 44

3.1 Average recognition rates on Caltech-101 with 30 training samples per class. Each row contains results for one of the training protocols (U = unsupervised, X = random, + = supervised fine-tuning), and each column for one type of architecture (F = filter bank, P_A = average pooling, P_M = max pooling, R = rectification, N = normalization). . . 63

3.2 CN_1: base model. N: Local Normalization layer (note: only the Y channel is normalized, U and V are untouched); C: convolutional layer; P: pooling (max) layer; L: linear classifier. 81

3.3 CN_2: second model. Filters are increased, which doubles the receptive field 83

3.4 CN_3: a fourth convolutional layer C6 is added, which, again, increases the receptive field. Note: C6 has sparse connectivity (*e.g.*each of its 128 outputs is connected to 8 inputs only, yielding 1024 kernels instead of 6144). 84

3.5 Percentage of mislabeled pixels on validation set. CN Error is the pixelwise error obtained when using the simplest pixelwise winner, predicted by the ConvNet. CN+MST Error is the pixelwise error obtained by histogramming the ConvNet's prediction into connected components (the components are obtained by computing the minimum spanning tree of an edge-weighted graph built on the raw RGB image, and merging its nodes using a surface criterion, in the spirit of (35)). 84

3.6 Performance comparison. 1- CPU: Intel DuoCore, 2.7GHz, optimized C code, 2- V6: neuFlow on Xilinx Virtex 6 FPGA—on board power and GOPs measurements; 3- IBM: neuFlow on IBM 45nm process: simulated results, the design was fully placed and routed; 4- mGPU/GPU: two GPU implementations, a low power GT335m and a high-end GTX480. . 85

1

Introduction

Central to this thesis is the question: how can we enable computers to automatically and efficiently understand images? *Understand* being an ambiguous term, we start with a few definitions. From the dictionary:

Definition 1 — understand:
(1) to perceive the meaning of; grasp the idea of; comprehend.
(2) to assign a meaning to; interpret.
(3) to have a systematic interpretation or rationale, as in a field or area of knowledge.

Definition 2 — image:
an optical counterpart or appearance of an object, as is produced by reflection from a mirror, refraction by a lens, or the passage of luminous rays through a small aperture and their reception on a surface.

From these we can provide our own definition:

Definition 3 — understand an image:
(1) to perceive the meaning behind the formation of the image
(2) to systematically interpret the causes—the physical objects and events—that resulted in the formation of the image.

This definition is not perfect, but it gives us a scope for this thesis. From this, it is easy to see how vast the task of understanding an image can be. Given the pixels, one would have to infer all the causes that led to this image: lighting conditions, exact list of all objects present in the receptive field, their exact 3D positions, contours, colors, surface normals...

In this thesis I focus on a subset of these explaining factors, which is commonly referred to as semantic labeling, or image parsing:

Definition 4 — parse an image:
given an image (an array of pixels), produce a 2D map (in the plane of the image) of objects, with their precise contour, position, and label (from a pre-defined label set).

The task of image parsing is significantly simpler than the task of full image understanding, and yet captures most of its fundamental problems: representation, recognition, segmentation...

The core of my thesis can be broken up into three main contributions:

1. a multiscale deep convolutional network architecture to easily capture long-distance relationships between input variables in image data. This type of model produces invariant yet spatially accurate features, which provide a good basis for image parsing,

2. a tree-based algorithm to efficiently explore multiple segmentation candidates, to produce maximally confident semantic segmentations of images. This type of method is computationally efficient, and provides a simple-to-use post-processing framework for image parsing,

3. a custom dataflow computer architecture optimized for the computation of convolutional networks, and similarly dense image processing models. This computer is fully implemented and functional.

The goal of this introduction is to put each contribution in perspective, and better understand where they come from, with one section per contribution. I start with a review of representation learning using deep networks. The second section provides context on the problem of structured prediction, and the use of segmentation trees. The third section describes dataflow computers, and why they are particularly well suited compute models for data-intensive tasks such as image parsing.

1.1 Representation Learning with Deep Networks

"Deep learning is just a buzzword for neural nets, and neural nets are just a stack of matrix-vector multiplications, interleaved with some non-linearities. No magic there."
— Ronan Collobert, 2011 (24)

One of the key questions of Vision Science (natural and artificial) is how to produce good internal representations of the visual world. What sort of internal representation would allow an artificial vision system to detect and classify objects into categories, independently of pose, scale, illumination, conformation, and clutter? More interestingly, how could an artificial vision system *learn* appropriate internal representations automatically, the way animals and humans seem to learn by simply looking at the world? In the time-honored approach to computer vision (and to pattern recognition in general), the question is avoided: internal representations are produced by a hand-crafted feature extractor, whose output is fed to a trainable classifier. While the issue of learning features has been a topic of interest for many years, considerable progress has been achieved in the last few years with the development of so-called *deep learning* methods.

Good internal representations are hierarchical. In vision, pixels are assembled into edglets, edglets into motifs, motifs into parts, parts into objects, and objects into scenes. This suggests that recognition architectures for vision (and for other modalities such as audio and natural language) should have multiple trainable stages stacked on top of each other, one for each level in the feature hierarchy. Deep neural networks are particularly well suited to represent hierarchical signals, as the overall function is naturally decomposed into a hierarchy of simpler, linear functions. Convolutional neural networks are an extension of deep neural networks, in which each layer imposes spatial (or temporal) replication of the weights, to exploit the stationarity and locality of the signal at each layer.

1.1.1 Deep Network Architectures

In this section I review well-known deep network architectures.

1.1.1.1 Multilayer Perceptrons

The first deep network, or deep learner, was the multilayer perceptron (MLP). An MLP typically consists of multiple layers of nodes arranged in a directed graph, with each layer fully connected to the next one. A node, or neuron at each layer is produced by a non-linear activation function of a linear combination of activations at the previous layer.

Mathematically, an MLP with L layers can be described by these simple equations:

$$\mathbf{y} = f(\mathbf{x};\theta) = \mathbf{h}_L, \tag{1.1}$$

$$\mathbf{h}_l = \mathrm{act}_l(\mathbf{W}_l\mathbf{h}_{l-1} + \mathbf{b}_l) \quad \forall \quad l \in \{1,\ldots,L-1\}, \tag{1.2}$$

$$\mathbf{h}_0 = \mathbf{x}, \tag{1.3}$$

with $\mathbf{b}_l$ a vector of trainable bias parameters, $\mathbf{W}_l$ a matrix of trainable weights, $\mathbf{x}$ is the input vector, $\mathbf{y}$ is a vector of output units, θ is a vector that represents all the trainable parameters $\{\mathbf{W}_l, \mathbf{b}_l\}$ $\forall$ $l \in \{1,\ldots,L\}$, and act_l is a non-linear activation function at layer l.

The most commonly used activation function for the hidden units act_l $\forall$ $l \in \{1,\ldots,L-1\}$) is tanh, but other more exotic transfer functions, such as the rectified linear unit (ReLU) can be used to effectively train deeper architectures. The output activation function act_L depends on the problem at hand. For regression problems, it can be a simple linear function, or a log-linear function. For discrimination problems, the softmax function is the most widely used, for its connection to maximum a posteriori probability (MAP) estimation. The softmax normalizes the output units so that they sum to 1, which turns the MLP into an approximator for the posterior probability $P(\mathbf{Y} = t^n|\mathbf{x}^n,\theta)$. When using a softmax activation function, the training procedure becomes analogous to MAP estimation in the sense that we seek the training parameter vector θ that maximizes the likelihood over all training samples $\{\mathbf{x}^n, t^n\}$.

Note: some textbooks consider the input vector x as a layer. In this thesis I only count the hidden layers and the output layer. This way, a simple linear model is considered a one-layer model, whereas the smallest MLP is considered a two-layer model (with one hidden layer). Effectively, I'm counting each linear projection as a layer.

1.1.1.2 Convolutional Networks

Many successful object recognition systems use dense features extracted on regularly-spaced patches over the input image. The majority of the feature extraction systems have a common structure composed of a filter bank (generally based on oriented edge detectors or 2D gabor functions), a non-linear operation (quantization, winner-take-all, sparsification, normalization, and/or point-wise saturation) and finally a pooling operation (max, average or histogramming). For example, the scale-invariant feature transform (SIFT (73)) operator applies oriented edge filters to a small patch and determines the dominant orientation through a winner-take-all operation. Finally, the resulting sparse vectors are added (pooled) over a larger patch to form local orientation histograms. Some recognition systems use a single stage of feature extractors (28, 60, 87). Other models like HMAX-type models (77, 93) and convolutional networks use two or more layers of successive feature extractors.

Put simply, Convolutional Networks (64, 65), or ConvNets are an extension of multilayer perceptrons, where the basic linear layers are replaced by convolutional layers. Non-linear activations are commonly followed by a spatial pooling function, which enforces low-level shift invariance.

Mathematically, a ConvNet with L layers can be described as an MLP, where we write the states as matrices (or more precisely arrays, or collections of vectors):

$$\mathbf{Y} = f(\mathbf{X};\theta) = \mathbf{H}_L, \tag{1.4}$$

$$\mathbf{H}_l = \text{pool}_l(\text{act}_l(\mathbf{W}_l\mathbf{H}_{l-1} + \mathbf{b}_l)) \quad \forall \quad l \in \{1,\ldots,L-1\}, \tag{1.5}$$

$$\mathbf{H}_0 = \mathbf{X}, \tag{1.6}$$

with $\mathbf{b}_l$ a vector of trainable bias parameters, $\mathbf{W}_l$ a matrix of trainable weights, $\mathbf{X}$ is the input array of vectors (an image is an array of pixels), $\mathbf{Y}$ is an array of output vectors (each vector encodes a sub-window of the input), θ is a vector that represents all the trainable parameters $\{\mathbf{W}_l, \mathbf{b}_l\}$ $\forall$ $l \in \{1,\ldots,L\}$, act_l is a non-linear activation function at layer l, and pool_l is a pooling function at layer l.

The major difference with the MLP is that the matrices $\mathbf{W}_l$ are Toeplitz matrices, therefore each hidden unit array $\mathbf{H}_l$ can be expressed as a regular convolution between kernels from $\mathbf{W}_l$ and the previous hidden unit vector $\mathbf{H}_{l-1}$, squashed through an act_l

function, and pooled spatially. More specifically,

$$\mathbf{H}_{lp} = \text{pool}(\text{act}(b_{lp} + \sum_{q \in \text{parents}(p)} \mathbf{w}_{lpq} * \mathbf{H}_{l-1,q})). \tag{1.7}$$

The hidden units $\mathbf{H}_l$ are commonly called *feature vector maps*, and $\mathbf{H}_{lp}$ is called a *feature map*. Concretely, if the input is a color image, each feature map would be a 2D array containing a color channel of the input image (for an audio input each feature map would be a 1D array, and for a video or volumetric image, it would be a 3D array). At the output, each feature map represents a particular feature extracted at all locations on the input.

From the mathematical description above, we can identify three key building blocks of ConvNets: the convolutional layer, or *filter bank layer*, the activation function, or *non-linearity layer*, and the pooling function, or *feature pooling layer*. A typical ConvNet is composed of one, two or more such 3-layer stages. The output of a ConvNet is usually fed into an simple linear classifier, or, more generally into an MLP. From the training/optimization point of view, the complete stack (ConvNet+MLP) can be treated as an MLP: for discriminative tasks, the usual softmax activation function is used as the output activation module, so that the optimization becomes a MAP estimation problem.

We now describe these three building blocks, which are used extensively throughout this thesis (see Figure 1.1):

Filter Bank Layer - F: the input is a 3D array with n_1 2D *feature maps* of size $n_2 \times n_3$. Each component is denoted x_{ijk}, and each feature map is denoted x_i. The output is also a 3D array, y composed of m_1 feature maps of size $m_2 \times m_3$. A trainable filter (kernel) k_{ij} in the filter bank has size $l_1 \times l_2$ and connects input feature map x_i to output feature map y_j. The module computes $y_j = b_j + \sum_i k_{ij} * x_i$ where $*$ is the 2D discrete convolution operator and b_j is a trainable bias parameter. Each filter detects a particular feature at every location on the input. Hence spatially translating the input of a feature detection layer will translate the output but leave it otherwise unchanged.

Non-Linearity Layer: In traditional ConvNets this simply consists in a pointwise tanh() sigmoid function applied to each site (ijk). However, recent implementations have used more sophisticated non-linearities. A useful one for natural image recognition is the rectified sigmoid R_{abs}: abs($g_i.tanh()$) where g_i is a trainable gain parameter. The

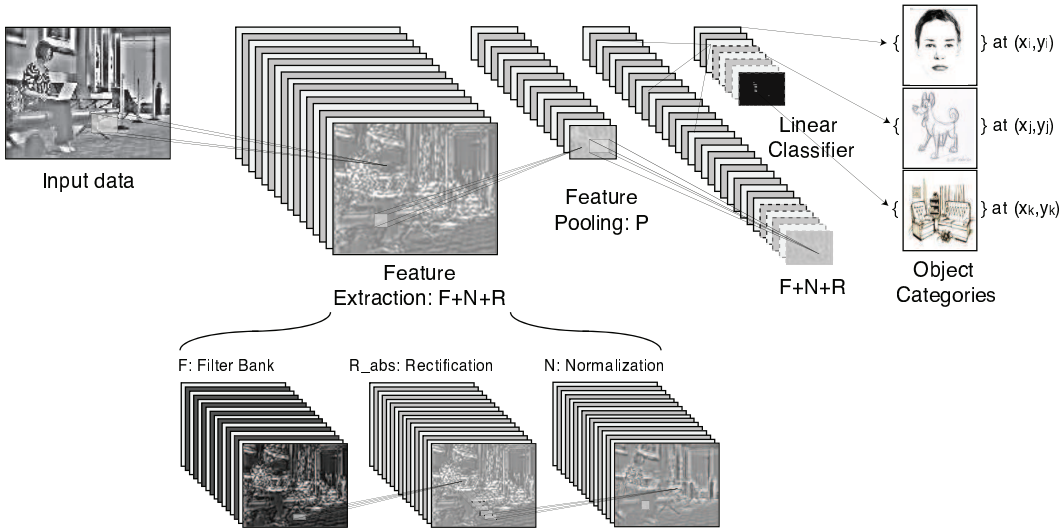

Figure 1.1: Common ConvNet Blocks - Architecture of a typical convolutional network for object recognition. This implements a convolutional feature extractor and a linear classifier for generic N-class object recognition. Once trained, the network can be computed on arbitrary large input images, producing a classification map as output.

rectified sigmoid is sometimes followed by a subtractive and divisive local normalization N, which enforces local competition between adjacent features in a feature map, and between features at the same spatial location. The subtractive normalization operation for a given site x_{ijk} computes: $v_{ijk} = x_{ijk} - \sum_{ipq} w_{pq}.x_{i,j+p,k+q}$, where w_{pq} is a normalized truncated Gaussian weighting window (typically of size 9x9). The divisive normalization computes $y_{ijk} = v_{ijk}/\max(mean(\sigma_{jk}), \sigma_{jk})$ where $\sigma_{jk} = (\sum_{ipq} w_{pq}.v^2_{i,j+p,k+q})^{1/2}$. The local contrast normalization layer is inspired by visual neuroscience models (74, 87).

Feature Pooling Layer: This layer treats each feature map separately. In its simplest instance, called P_A, it computes the average values over a neighborhood in each feature map. The neighborhoods are stepped by a stride larger than 1 (but smaller than or equal to the pooling neighborhood). This results in a reduced-resolution output feature map which is robust to small variations in the location of features in the previous layer. The average operation is sometimes replaced by a max P_M. Traditional ConvNets use a pointwise $tanh()$ after the pooling layer, but more recent models do not. Some ConvNets dispense with the separate pooling layer entirely, but use strides larger than one in the filter bank layer to reduce the resolution (63, 96). In some recent

versions of ConvNets, the pooling also pools different features at a same location, in addition to the same feature at nearby locations (54).

(A Short History of ConvNets)

ConvNets can be seen as a representatives of a wide class of models that we will call *Multi-Stage Hubel-Wiesel Architectures*. The idea is rooted in Hubel and Wiesel's classic 1962 work on the cat's primary visual cortex. It identified orientation-selective *simple cells* with local receptive fields, whose role is similar to the ConvNets filter bank layers, and *complex cells*, whose role is similar to the pooling layers. The first such model to be simulated on a computer was Fukushima's Neocognitron (38), which used a layer-wise, unsupervised competitive learning algorithm for the filter banks, and a separately-trained supervised linear classifier for the output layer. The innovation in (63, 64) was to simplify the architecture and to use the back-propagation algorithm to train the entire system in a supervised fashion. The approach was very successful for such tasks as OCR and handwriting recognition. An operational bank check reading system built around ConvNets was developed at AT&T in the early 1990's (65). It was first deployed commercially in 1993, running on a DSP board in check-reading ATM machines in Europe and the US, and was deployed in large bank check reading machines in 1996. By the late 90's it was reading over 10% of all the checks in the US. This motivated Microsoft to deploy ConvNets in a number of OCR and handwriting recognition systems (18, 19, 96) including for Arabic (1) and Chinese characters (17). Supervised ConvNets have also been used for object detection in images, including faces with record accuracy and real-time performance (40, 80, 84, 101), Google recently deployed a ConvNet to detect faces and license plate in StreetView images so as to protect privacy (37). NEC has deployed ConvNet-based system in Japan for tracking customers in supermarket and recognizing their gender and age. Vidient Technologies has developed a ConvNet-based video surveillance system deployed in several airports in the US. France Télécom has deployed ConvNet-based face detection systems for video-conference and other systems (40). Other experimental detection applications include hands/gesture (82), logos and text (29). A big advantage of ConvNets for detection is their computational efficiency: even though the system is trained on small windows, it suffices to extend the convolutions to the size of the input image and replicate the output layer to compute detections at every location. Supervised ConvNets have also been used for vision-based obstacle avoidance for off-road mobile robots (67). Two participants

in the recent DARPA-sponsored LAGR program on vision-based navigation for off-road robots used ConvNets for long-range obstacle detection (45, 46). In (45), the system is pre-trained off-line using a combination of unsupervised learning (as described in section 3.2.2) and supervised learning. It is then adapted on-line, as the robot runs, using labels provided by a short-range stereovision system (see videos at `http://www.cs.nyu.edu/~yann/research/lagr`). Interesting new applications include image restoration (50) and image segmentation, particularly for biological images (81). The big advantage over graphical models is the ability to take a large context window into account. Stunning results were obtained at MIT for reconstructing neuronal circuits from a stack of brain slice images a few nanometer thick (51).

Over the years, other instances of the Multi-Stage Hubel-Wiesel Architecture have appeared that are in the tradition of the Neocognitron: unlike supervised ConvNets, they use a combination of hand-crafting, and simple unsupervised methods to design the filter banks. Notable examples include Mozer's visual models (75), and the so-called HMAX family of models from T. Poggio's lab at MIT (77, 93), which uses hard-wired Gabor filters in the first stage, and a simple unsupervised random template selection algorithm for the second stage. All stages use point-wise non-linearities and max pooling. From the same institute, Pinto et al. (87) have identified the most appropriate non-linearities and normalizations by running systematic experiments with a single-stage architecture using GPU-based parallel hardware.

1.1.1.3 Encoders + Decoders = Auto-encoders

Training deep, multi-stage architectures using supervised gradient back propagation requires many labeled samples. However in many problems labeled data is scarce whereas unlabeled data is abundant. Recent research in deep learning (7, 48, 88) has shown that *unsupervised learning* can be used to train each stage one after the other using only unlabeled data, reducing the requirement for labeled samples significantly.

Learning features in an unsupervised manner (*i.e.* without labels) can be achieved simply, by using auto-encoders. An auto-encoder is a model that takes a vector input $\mathbf{y}$, maps it into a hidden representation $\mathbf{z}$ (code) using an encoder which typically has the form:

$$\mathbf{z} = \mathrm{act}(\mathbf{W}_e\mathbf{y} + \mathbf{b}_e), \tag{1.8}$$

where act is a non-linear activation function, $\mathbf{W}_e$ the encoding matrix and $\mathbf{b}_e$ a vector of bias parameters.

The hidden representation $\mathbf{z}$, often called code, is then mapped back into the space of $\mathbf{y}$, using a decoder of this form:

$$\tilde{\mathbf{y}} = \mathbf{W}_d \mathbf{z} + \mathbf{b}_z, \tag{1.9}$$

where $\mathbf{W}_d$ is the decoding matrix and $\mathbf{b}_d$ a vector of bias parameters.

The goal of the auto-encoder is to minimize the reconstruction error, which is represented by a distance between $\mathbf{y}$ and $\tilde{\mathbf{y}}$. The most common type of distance is the mean squared error $||\mathbf{y} - \tilde{\mathbf{y}}||_2^2$.

The code $\mathbf{z}$ typically has less dimensions than $\mathbf{y}$, which forces the auto-encoder to learn a good representation of the data. In its simplest form (linear), an auto-encoder learns to project the data onto its first principal components. If the code $\mathbf{z}$ has as many components as $\mathbf{y}$, then no compression is required, and the model could typically end up learning the identity function. Now if the encoder has a non-linear form (using a tanh, or using a multi-layered model), then the auto-encoder can learn a potentially more powerful representation of the data.

Basic auto-encoders require a number of tricks and *know how* to properly train them, and avoid the pitfall of learning the identity function. In practice, using a code $\mathbf{y}$ that is smaller than $\mathbf{x}$ is enough to avoid learning the identity, but it remains hard to do much better than PCA. Techniques like the denoising auto-encoder (DAE), introduced in (102) can be useful to avoid that.

Using codes that are over-complete (*i.e.* with more components than the input) makes the problem even worse. There are different ways that an auto-encoder with an over-complete code may still discover interesting representations. One common way is the addition of sparsity: by forcing units of the hidden representation to be mostly 0s, the auto-encoder has to learn a distributed representation of the data. More advanced methods, such as Predictive Sparse Coding (PSD) (53), involve learning an encoder that approximates the exact result of sparse coding. Sparse Coding can be a bit costly, as it is an iterative procedure, whereas the encoder will predict the sparse code in a feedforward way.

The auto-encoder loss can be used by itself for purely unsupervised pre-training. The parameters are then used to initialize the supervised procedure. It can also be

used in conjunction with the supervised training, to ensure that there is no loss of information at each layer: if the auto-encoder loss is perfectly minimized, it means that the top layer representation contains all the information required to rebuild the input signal. This can be useful for tasks where certain labels have too few training examples, such that it is dangerous to rely on the label information alone.

1.1.2 Learning: Parameter Estimation

In this thesis I focus on deep networks for discriminative tasks. Therefore, I will only consider learning (parameter estimation) for discriminative tasks.

1.1.2.1 Loss Function, Objective

From the point of view of parameter estimation, the architecture of the model can usually be abstracted. In the following, we assume a training set of N training samples $\{\mathbf{x}^n, t^n\}$, with $\mathbf{x}^n$ an input example, and t^n a target value, or label, associated to that example; $t^n \in \{1, \ldots, K\}$, with K the number of possible target classes. We can write:

$$\mathbf{y}^n = f(\mathbf{x}^n; \theta) \quad \forall \quad n \in \{1, \ldots, N\}, \tag{1.10}$$

$$l(f; \mathbf{x}^n, t^n, \theta) = l(f(\mathbf{x}^n; \theta), t^n) \quad \forall \quad n \in \{1, \ldots, N\} \tag{1.11}$$

$$L(f; \mathbf{x}, t, \theta) = \sum_{n \in \{1, \ldots, N\}} l(f; \mathbf{x}^n, t^n, \theta) \tag{1.12}$$

where f is a model with trainable parameters θ, l is a loss function which captures the per-sample objective to be optimized, and L the global loss function which represents the overall objective to be optimized.

As described in Section 1.1.1, the use of a softmax output activation function allows us to turn the learning problem into a likelihood maximization problem, or negative log-likelihood minimization problem, which gives:

$$l(f(\mathbf{x}^n; \theta), t^n) = -\log(P(Y = t^n | \mathbf{x}^n, \theta)) \tag{1.13}$$

$$= -\log(f(\mathbf{x}^n)). \tag{1.14}$$

There are several other types of possible loss functions, but the negative log-likelihood (NLL) provides a simple and consistent parameter estimation framework, in which the outputs of f are properly calibrated units.

1.1.2.2 Optimization

Once a model f and a loss function l have been chosen, we can define the task of learning, or parameter estimation, as minimizing the loss function L over the training set $\{\mathbf{x}^n, t^n\}\ \forall\ n \in \{1, \ldots, N\}$. If f and l are differentiable, or at least piece-wise differentiable, this optimization can be cast as a gradient descent procedure.

The most naive way to go about solving this optimization problem is to compute the derivative of the loss function with respect to all the trainable parameters (using the well-known *backpropagation* algorithm), over the complete training set, and then follow the opposite direction to update the parameters. It is naive for two reasons: (1) it only relies on first order information (the gradient), (2) it relies on the entire dataset to evaluate the gradient (full batch), which is typically extremely inefficient.

The first point can be addressed using parameter normalization, hidden unit normalization, (partial) second-order information... Different types of normalizations are presented throughout this thesis.

The second point is typically addressed using a stochastic approximation of the gradient, usually referred to as stochastic gradient descent (SGD). The most extreme form of SGD is when a single sample $\{\mathbf{x}^n, t^n\}$ is used to estimate the gradient, and to update the parameters. We usually use the term mini-batch to describe the set of samples used to evaluate the gradient and update the parameters. The mini-batch size can vary from 1 (pure SGD) to N (batch method, or exact method).

All the algorithms presented in this thesis rely on some form of stochasticity. Several studies (11, 12, 13) have shown that even when f is a convex function with respect to the trainable parameters, SGD yields significantly faster convergence, and when combined to a proper learning rate schedule and/or validation scheme, reaches the same accuracy as exact methods. SGD was used extensively in this thesis.

1.2 Hierarchical Segmentations, Structured Prediction

The focus of this thesis is on image understanding, and more precisely on image parsing, or multi-label segmentation. Although it is theoretically doable to build a deep model f which can remap a raw image signal $\mathbf{X}$ into a map of discrete labels, the use of heuristics—candidate segmentations—can greatly speedup the learning process, and the overall consistency of image labelings.

In this section I provide material and context for Chapter 2. I start with an introduction on hierarchical segmentations, which are used throughout Chapter 2. I then present the general ideas of structured prediction, a good paradigm for sequence data and spatial data labeling problems.

1.2.1 Hierarchical Segmentations

An image segmentation is a partitioning of an image into regions corresponding to different objects. A hierarchical image segmentation is an ensemble of image segmentations where the image segments are arranged in a tree-like structure. The root of the tree is a single segment that spans all the pixels of the image, and the leaves of the tree are the individual pixels (one component per pixel). This type of data structure is particularly useful to explore different levels of candidate segmentations.

In this section, I describe the basics required to build graphs on images, and produce hierarchical segmentations on these graphs.

1.2.1.1 Graph Representation

A graph G is defined by a set of vertices V and a set of edges E that connect the vertices. In this thesis, we use the convention of edge-weighted, undirected graphs, to represent images: a pixel is represented by a vertex, and a link between two pixels is represented by a weighted edge.

A complete graph over an image is defined when each pixel is connected to every other pixel in the image. Such graph is typically very costly to represent in memory, as its number of edges scales quadratically with the number of pixels.

A much more common type of graph is locally connected: each pixel is connected to its most immediate 4 neighbors (4-connexity) or its 8 immediate neighbors (8-connexity), as shown on Figure 1.2.

A very simple and natural kind of graph is a gradient graph: such a graph can be built by setting the connexity to 4, and assigning each edge a weight that is the Euclidean distance between its two neighboring vertices. This graph represents a gradient map: each edge encodes a distance between pixels, as shown on Figure 1.3.

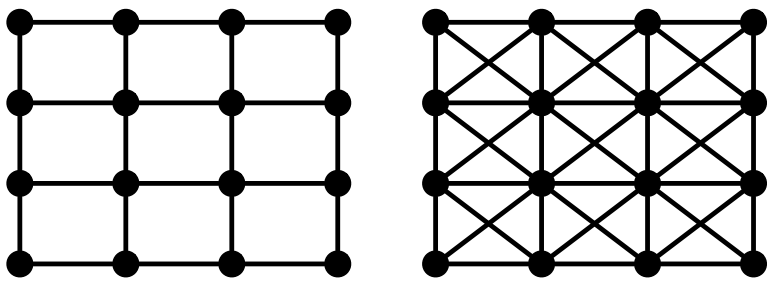

Figure 1.2: Graphs with local connectivity - Left: 4-connexity. Right: 8-connexity.

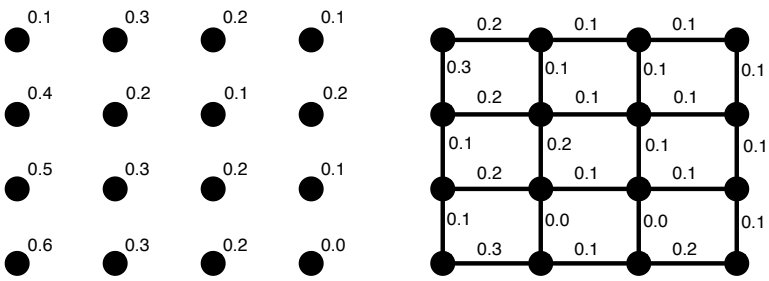

Figure 1.3: Gradient graph - This type of graph is edge-weighted. Left: vertices, with weights attached. Right: the edge-weighted gradient graph—each edge has a weight associated, which is produced by the distance between its two neighboring vertices.

1.2.1.2 Minimum Spanning Trees

Once a graph is constructed over an image, we can start thinking about trees. A spanning tree of a graph G is itself a graph T, that contains all the vertices of G but a subset of the edges in G that span all the vertices. For a given graph G over an image, there are multiple possible spanning trees. A minimum spanning tree T_{MST} of an edge-weighted graph G is the subset of edges chosen such that they minimize the sum of the edge weights. A key property of spanning trees is that they contain no loops (which is why they are called trees!).

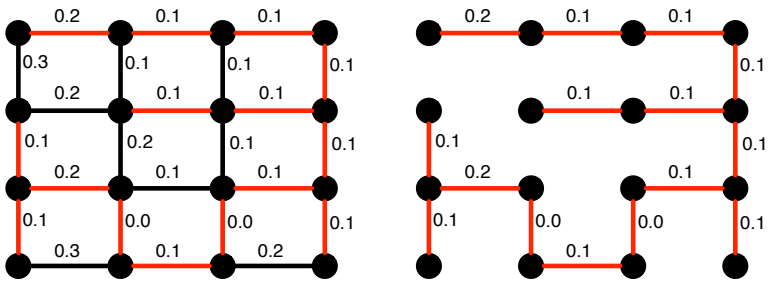

Figure 1.4: Minimum Spanning Tree of a Graph - Left: highlighting a possible MST for the graph in Figure 1.2. Right: pruning the graph to only keep the edges belonging to the MST—these edges cover the graph.

There are multiple well-known algorithms for finding MSTs. One of them is Kruskal's algorithm (56, 79), which constructs the MST by sorting all the edges by increasing weight, and adds them one by one if they do not create cycles. The algorithm maintains a list of clusters, and ensures that each time it adds an edge to the MST, the edge fuses two distinct clusters (if the two neighboring vertices already belonged to the same cluster, then adding that extra edge would create a cycle). Thus the only challenge of the Kruskal algorithm is to efficiently keep track of the clusters. Using disjoint sets and path compression, the overall complexity of the algorithm can be kept to $O(|E|.\alpha(|E|))$, where $|E|$ is the number of edges in the graph, and $\alpha(.)$ is the inverse Ackermann function, a function that grows very slowly with its argument. In other terms, the Kruskal algorithm is roughly linear in the number of edges, when correctly implemented.

This is an important conclusion, as it tells us that we can compute minimum spanning trees very cheaply.

1.2.1.3 Dendograms

A minimum spanning tree is an efficient data structure to access a grid of pixels, and have them organized by increasing edge weights. But the minimum spanning tree is in fact more informative: it actually captures a full segmentation hierarchy. To see that, the spanning tree must be visualized using a dendogram. A dendogram is a rooted binary tree whose leaf nodes consist of the objects being clustered, in our case the pixels (vertices of the graph). Each internal node of the dendogram represents a cluster corresponding to all its child leaf nodes. These nodes have a one-to-one correspondance with the edges of the MST, and the height of each internal node represents the weight of the edge in the MST! See Figure 1.5.

Concretely, looking at the dendogram of a gradient graph built on an image shows: (1) high nodes corresponding to strong edges in the image, and (2) low nodes corresponding to flat areas / edge-free areas in the image.

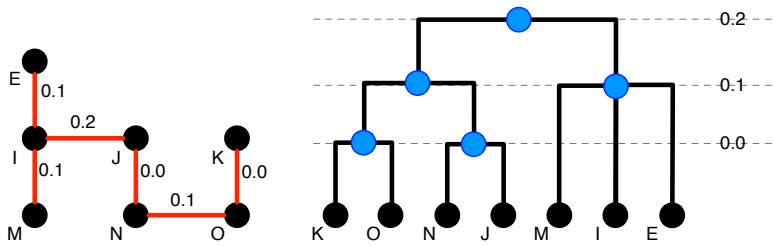

Figure 1.5: Dendogram of an MST - Left: a subset of the MST in Figure 1.4. Right: its dendogram. The bottom nodes are the vertices in the original graph; the blue nodes represent merging levels.

1.2.1.4 Segmentations

Given a dendogram, a single segmentation of the image can be obtained very easily, by cutting the dendogram at a fixed altitude, or threshold. After cutting the dendogram at a fixed altitude, we are left with a set of subtrees, called connected components. These connected components cover the original image (*i.e.* each node in the original graph belongs to one and only one component), and represent a possible segmentation of the image. Intuitively, if the original edge weighting function is a simple Euclidean distance between neighbors, that segmentation is very brittle, as it depends on very local edge information.

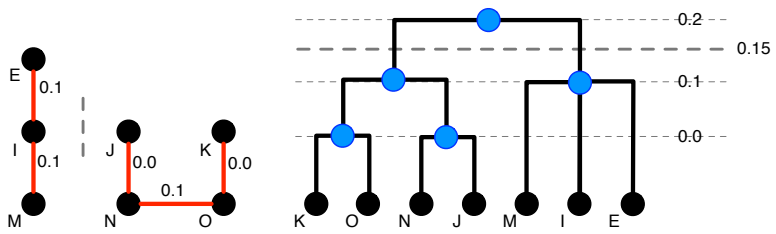

Figure 1.6: Cutting the Dendogram = Segmenting - Left: two connected components, obtained after thresholding at 0.15. Right: dendogram of MST, any node above the threshold will be removed. the graph.

Alternatively, the dendogram can be filtered, according to criterions that depend on the geometry of the underlying component (surface, volume, ...). Felzenszwalb & Huttenlocher (35) proposed an interesting method to produce final components, using a criterion that compares the maximum weight within a component to each edge between any two components, and have an adaptive threshold that depends on this ratio. The method produces balanced segmentations which are robust to local noise. This type of technique effectively performs a non-horizontal cut of the hierarchy, taking into account the local morphology of each subtree.

More advanced forms of thresholding criteria can even involve learning. This can be achieved by defining a function over a neighborhood of pixels, to produce the edge costs in such a way that graph-cut segmentation and similar methods produce the best answer. One such objective function is Turaga's Maximin Learning (100), which pushes up the lowest edge cost along the shortest path between two points in different segments, and pushes down the highest edge cost along a path between two points in the same segment.

In this thesis I was mostly interested in using segmentation hierarchies to explore a large set of candidate segmentations in an efficient way. The focus of the thesis is thus more on the use of such trees, as complements to feature learners, rather than on their production.

1.2.2 Structured Prediction

Structured prediction is a term that describes techniques that involve predicting structured objects, *i.e.*considering output labels as inter-dependent, and explicitly modeling this inter-dependence. One of the earliest structured prediction systems was proposed

by LeCun *et al.* (65), to address the problem of labeling scanned documents. The challenge there is that there is a joint problem of segmentation and recognition: given a long bitmap of characters, one must jointly find the best segmentation into characters and classify each character. Any problem that involves labeling sequence data can be treated with the technique proposed in (65): speech recognition, natural language processing, handwritten text recognition (OCR), music transcription...

For sequence data, one can easily produce all possible segmentations, and compute a unary cost for each segment. The Viterbi algorithm can then be used to find the most likely sequence of labels.

For image labeling problems, segmentation becomes problematic, as there is no way to exhaustively explore all possible segmentation candidates: the graph being loopy, decoding has to be done in an approximate way, using techniques like loopy belief-propagation, or graph cuts.

1.2.2.1 Graphical Models

Let us start with a general introduction of undirected graphical models for structured prediction tasks. We assume a graph $G = (V, E)$ with vertices $i \in V$ and edges $e \in E \subseteq V \times V$. The joint probability of a particular assignment to all the variables x_i is represented as a normalized product of a set of non-negative potential functions:

$$p(x_1, x_2, \ldots, x_{|V|}) = \frac{1}{Z} \prod_{i \in V} \phi_i(x_i) \prod_{e_{jk} \in E} \phi_{e_{jk}}(x_j, x_k). \tag{1.15}$$

There is one node potential function ϕ_i for each node i, and one edge potential ϕ_e for each edge e. Each edge connects two nodes, e_j and e_k. In a complete graph, there is one edge between each possible pair of nodes. For common applications, such as computer vision, it's much more common to have locally connected graphs, *i.e.* graphs in which only (small) subsets of nodes are connected via edges (see previous section).

The node potential function ϕ_i gives a non-negative weight to each possible value of the random variable x_i. For example, we might set $\phi_i(x_i = 0)$ to 0.75 and $\phi_i(x_i = 1)$ to 0.25, which means means that node i has a higher potential of being in state 0 than state 1. Similarly, the edge potential function $\phi_{e_{jk}}$ gives a non-negative weight to all the combinations that x_j and x_k can take.

The normalization constant Z, or partition function, is a scalar value that forces the distribution to sum to one, over all possible joint configurations of the variables:

$$Z = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_{|V|}} \prod_{i \in V} \phi_i(x_i) \prod_{e_{jk} \in E} \phi_{e_{jk}}(x_j, x_k). \tag{1.16}$$

This normalizing constant ensures that the model defines a valid probability distribution.

Given a graph G, there are three tasks that are commonly performed:

- parameter estimation (learning): the task of computing the potential functions ϕ that maximize the likelihood of the training data (or, given a predefined function ϕ parametrized by $\mathbf{W}$, finding the optimal parameters $\mathbf{W}$);
- inference: the task of estimating the partition function Z as well as the marginal probabilities of each node taking each possible state;
- decoding: the task of finding the most likely joint configuration of the variables (the configuration that has the highest joint probability).

1.2.2.2 Learning: Parameter Estimation

As explained in Section 1.1, if we are only interested in discrimination (classification), Graphical models can be simplified by considering the negative log likelihood, and ignoring the normalization constant Z (which quickly becomes intractable and/or meaningless for large problems). We define the energy $E \propto -\log(p)$:

$$E(x_1, x_2, \ldots, x_{|V|}) = \sum_{i \in V} \Phi_i(x_i) + \sum_{e_{jk} \in E} \Phi_{e_{jk}}(x_j, x_k). \tag{1.17}$$

We assume that Φ_i and Φ_e are predefined functions (a linear model, a multilayer perceptron, or a convolutional network), parametrized by a set of trainable weights $\mathbf{w}$. For stationary data (images, audio...), it is common to have models that are fixed across locations, and that only depend on their input x_i and the groundtruth label t_i. Since they are constant across locations, we can drop the subscripts i and e, and rename them ϕ and ψ, which are now functions of x_i, t_i and $\mathbf{w}$. We can rewrite the energy as:

$$E(\mathbf{x}, \mathbf{t}; \mathbf{w}) = \sum_{i \in V} \Phi(x_i, t_i; \mathbf{w}_\phi) + \sum_{e_{jk} \in E} \Psi_{e_{jk}}(x_j, x_k, t_j, t_k; \mathbf{w}_\psi), \tag{1.18}$$

where $\mathbf{x}$ is the vector of input nodes, and $\mathbf{t}$ is the vector of groundtruth labels for each node. This energy is also known as the Conditional Random Field (CRF) energy.

The parameter estimation task (learning) becomes a simple minimization problem, similar to that described in Section 1.1. Reusing the same formulation, and assuming a training set of pairs $\{x^n, y^n\}\ \forall\ n \in \{1, \ldots, N\}$, we have:

$$l(\phi, \psi; \mathbf{x}^n, \mathbf{t}^n, \mathbf{w}) = E(\mathbf{x^n}, \mathbf{t}^n; \mathbf{w}) \quad \forall \quad n \in \{1, \ldots, N\} \tag{1.19}$$

$$L(\phi, \psi; \mathbf{x}, \mathbf{t}, \mathbf{w}) = \sum_{n \in \{1,\ldots,N\}} l(f; \mathbf{x}^n, \mathbf{t}^n, \mathbf{w}). \tag{1.20}$$

Depending on the forms of ϕ and ψ, the overall objective L might be convex or not. In the classical CRF literature, the potential functions are usually linear in their parameters, so the overall problem is indeed convex. Optimization details presented in Section 1.1.2.2 also apply here. In particular, stochastically estimating the gradients can tremendously accelerate the learning, as opposed to using more exact methods like L-BFGS.

More generally, the potential functions can be arbitrarily complex non-convex functions, for example, in the case of image labeling they could be full-blown convolutional networks, which depend on a neighborhood of input variables. In this case, the overall objective function becomes non-convex, and the learning problem challenging. A simpler solution is to modularize the process of learning, and do it in two steps: (1) train the unary potentials (the convolutional network) on individual input samples; (2) freeze the unary potential functions, pre-compute them for all images, and learn the CRF parameters (a convex problem). That second approach is the basis of Chapter 2.

One of the central results of this thesis is the fact that using a powerful node potential, such as a multiscale convolutional network (as presented in Chapter 2), can greatly reduce the need for a top down, global CRF, as each node potential manages to learn the structure of a large set of input variables.

1.3 Dataflow Computing

The third contribution of this thesis is a custom dataflow computer architecture optimized for the computation of convolutional networks (such as the model presented in Section 2). Dataflow computers are a particular type of processing architecture, which aim at maximizing the number of effective operations per instruction, which in

turn maximizes the number of operations reachable per second and per watt consumed. They are particularly well fit to the computation of convolutional networks, as these require very little branching logic, and rather require tremendous quantities of basic, redundant arithmetic operations.

In this section I provide a very quick primer on dataflow computing and architectures. As I suspect most readers of this thesis will come from a software background, this section is rather high-level, with an emphasis on the compute model rather than on the specific details of implementation. Chapter 3 extensively describes our custom dataflow architecture.

Dataflow architectures are a particular type of computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or (at least conceptually) the execution of instructions is solely determined based on the availability of input data to the compute elements.

Dataflow architectures have been successfully implemented in specialized hardware such as in digital signal processing (3, 85), network routing (5), graphics processing (71, 92). It is also very relevant in many software architectures today including database engine designs and parallel computing frameworks (9, 10).

Before getting into the details of the dataflow architecture, let us look at the Von Neumann architecture, which should help highlight the fundamental shortcomings of traditional flow control for highly data-driven applications.

In this type of architecture, the control unit, which decodes the instructions and executes them, is the central point of the system. A program (sequence of instructions) is typically stored in external memory, and sequentially read into the control unit. Certain types of instructions involve branching, while others involve reading data from the memory into the arithmetic logic unit (ALU), to transform them, and write them back into external memory.

When executing programs that are highly unpredictable in terms of branching (programs that have many possible execution paths, with an essentially uniform probability distribution), this type of architecture is optimal. The control unit loads one instruction per clock cycle, which either: (1) reads data into the ALU, (2) writes it back to memory, (3) triggers an ALU operation on data that are already in local registers, or

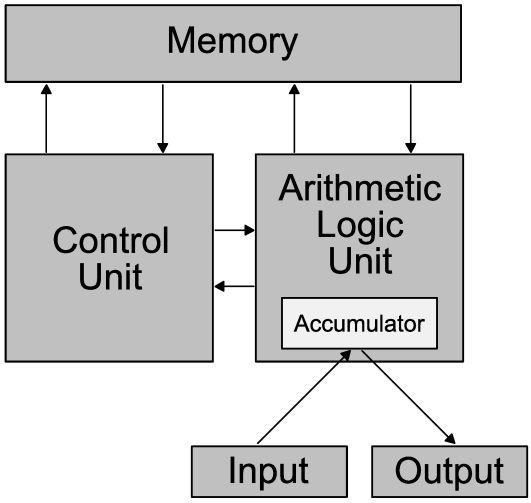

Figure 1.7: Von Neumann Architecture - Diagram of the Von Neumann architecture.

(4) does a branching (goto/jump). The level of granularity in the control flow is essentially maximal. At any time, between any two simplest operations can the program branch out and follow a new execution path.

The fact that control flow is so natural in Von Neumann architectures explains their wide success at implementing and running complex multi-tasking programs (OSes, I/O driven and asynchronous applications, ...).

In today's Von Neumann architectures, the control unit and ALU's clock speed run significantly faster than the external memory bus. It is largely compensated by the width of these buses, to provide sufficient bandwidth to sustain operations within the ALU, but this does not solve the problem of latency, which means that fetching a new item from memory results in potentially hundreds of idle CPU cycles. State-of-the-art control flow architectures (current X86/Intel processors) typically allocate more logic and therefore consumed power in caching + smart flow-control than in actual compute logic. This is required, as the applications running are completely driven by the control unit, which initiates and drive every atomic operation at a fine grained level. Great for flow control, terrible for data intensive applications.

This is where the dataflow model of computing comes in. In the ideal model, the general idea is to get rid of the control unit, and simply push streams of data into the

compute logic. When the data streams into the compute logic, computations occur continuously, and the processed data can be saved back into memory. There is no fine grained, cycle-accurate control, rather the flows of data themselves trigger the computations.

The core of the architecture proposed in this thesis relies on this idea of data-driven computations, complemented by a powerful online hardware re-configuration system, and a global, macroscopic control flow unit. Figure 1.8 provides an overview of this architecture.

The most striking aspect of this architecture is the ratio between actual compute logic and control+caching logic. Caching is essentially nil, as the entire architecture is designed to work on streams: as the streams produce the computations, there is no need for caching (there is no latency to hide, as the control unit works asynchronously). The control logic is very sparse in its activity, as it is only here to reconfigure the grid of processing tiles (PTs on the figure): before scheduling any new computation, it configures multiple tiles to perform given operations, and it also configures all the routes/connections between tiles and global data paths. Once the grid is configured, streams of data can come into it, and produce thousands, or millions of results, before a new configuration is required. Configurations can be initiated in parallel with the computations.

The Processing Tiles (PTs) are passive computers. Each tile can be configured to do one of several basic arithmetic tasks (including common DSP functions, like dot products, and convolutions).

Chapter 3 provides a full description of this system.

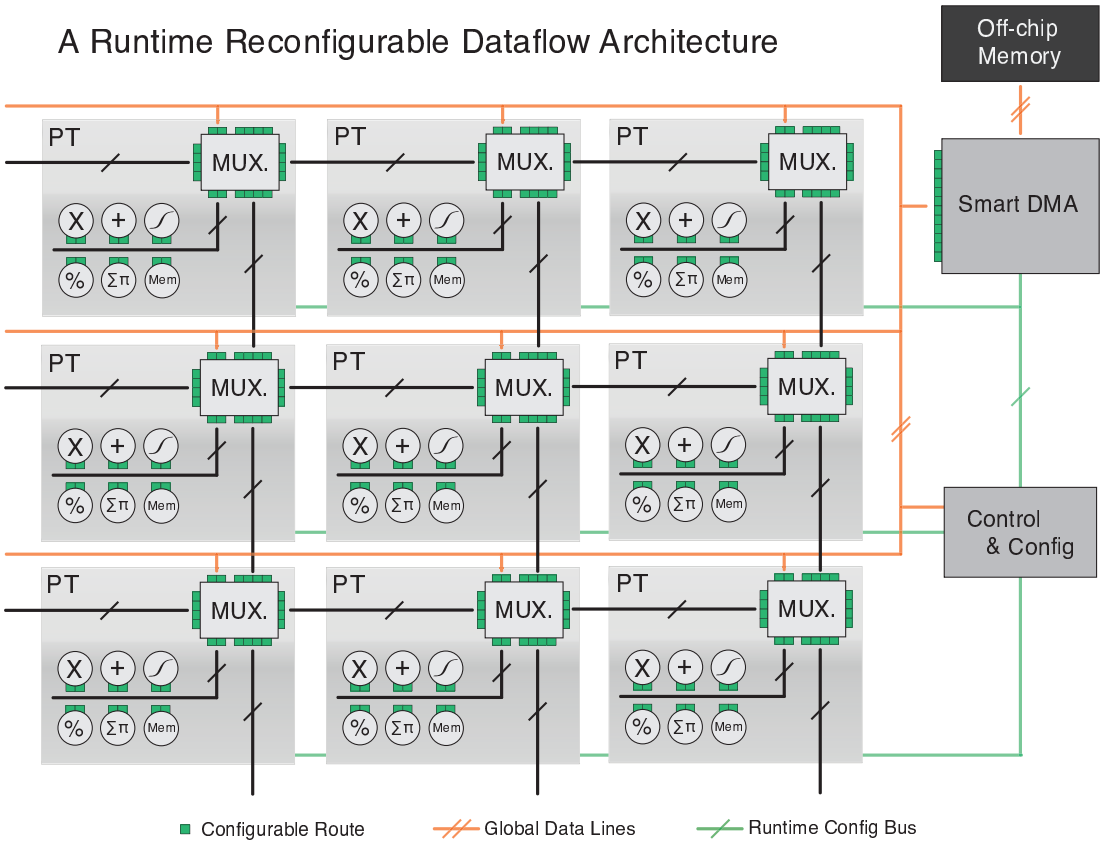

Figure 1.8: Our Proposed Dataflow Architecture - Diagram of our dataflow architecture.

2

Image Understanding: Scene Parsing

2.1 Introduction

Image understanding is a task of primary importance for a wide range of practical applications. One important step towards understanding an image is to perform a *full-scene labeling* also known as a *scene parsing*, which consists in labeling every pixel in the image with the category of the object it belongs to. After a perfect scene parsing, every region and every object is delineated and tagged. One challenge of scene parsing is that it combines the traditional problems of detection, segmentation, and multi-label recognition in a single process.

There are two questions of primary importance in the context of scene parsing: how to produce good internal representations of the visual information, and how to use contextual information to ensure the self-consistency of the interpretation.

2.2 A Model for Scene Understanding

2.2.1 Introduction

This chapter presents a scene parsing system that relies on deep learning methods to approach both questions. The main idea is to use a convolutional network (65) operating on a large input window to produce label hypotheses for each pixel location. The convolutional net is fed with raw image pixels (after band-pass filtering and contrast

normalization), and trained in supervised mode from fully-labeled images to produce a category for each pixel location. Convolutional networks are composed of multiple stages each of which contains a filter bank module, a non-linearity, and a spatial pooling module. With end-to-end training, convolutional networks can automatically learn hierarchical feature representations.

Unfortunately, labeling each pixel by looking at a small region around it is difficult. The category of a pixel may depend on relatively short-range information (*e.g.* the presence of a human face generally indicates the presence of a human body nearby), but may also depend on long-range information. For example, identifying a grey pixel as belonging to a road, a sidewalk, a gray car, a concrete building, or a cloudy sky requires a wide contextual window that shows enough of the surroundings to make an informed decision. To address this problem, we propose to use a *multi-scale convolutional network*, which can take into account large input windows, while keeping the number of free parameters to a minimum.

Common approaches to scene parsing first produce segmentation hypotheses using graph-based methods. Candidate segments are then encoded using engineered features. Finally, a conditional random field (or some other type of graphical model), is trained to produce labels for each candidate segment, and to ensure that the labelings are globally consistent.

A striking characteristic of the system proposed here is that the use of a large contextual window to label pixels reduces the requirement for sophisticated post-processing methods that ensure the consistency of the labeling.

More precisely, the proposed scene parsing architecture is depicted on Figure 2.1. It relies on two main components:

1) Multi-scale, convolutional representation: our multi-scale, dense feature extractor produces a series of feature vectors for regions of multiple sizes centered around every pixel in the image, covering a large context. The multi-scale convolutional net contains multiple copies of a simple network (all sharing the same weights) that are applied to different scales of a Laplacian pyramid version of the input image. For each pixel, the networks collectively encode the information present in a large contextual window around the given pixel (184×184 pixels in the system described here). The convolutional network is fed with raw pixels and trained end to end, thereby alleviating

the need for hand-engineered features. When properly trained, these features produce a representation that captures texture, shape and contextual information. While using a multiscale representation seems natural for scene parsing, it has rarely been used in the context of feature learning systems. The multiscale representation that is learned is sufficiently complete to allow the detection and recognition of all the objects and regions in the scene. However, it does not accurately pinpoint the boundaries of the regions, and requires some post-processing to yield cleanly delineated predictions.

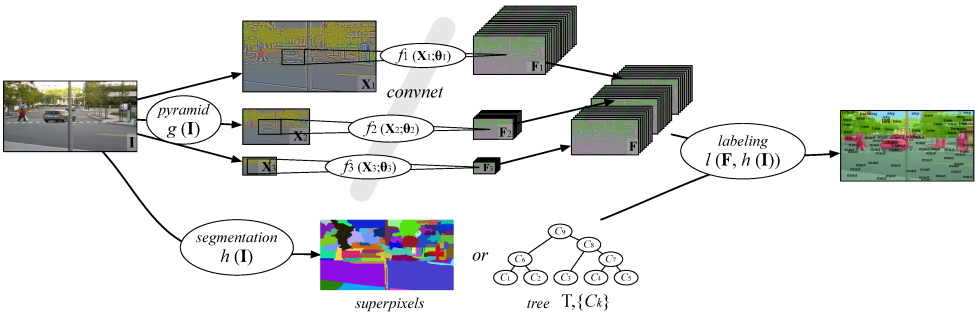

Figure 2.1: Model overview - Diagram of the scene parsing system. The raw input image is transformed through a Laplacian pyramid. Each scale is fed to a 3-stage convolutional network, which produces a set of feature maps. The feature maps of all scales are concatenated after having been upsampled to to match the size of the finest-scale map. Each feature vector thus represents a large contextual window around each pixel. In parallel, a single segmentation (*i.e.* superpixels), or a family of segmentations (*e.g.* a segmentation tree) are computed to exploit the natural contours of the image. The final labeling is produced from the feature vectors and the segmentation(s) using different methods, as presented in section 2.2.3.

2) Graph-based classification:

An over-segmentation is constructed from the image, and is used to group the feature descriptors. Several over-segmentations are considered, and three techniques are proposed to produce the final image labeling.

2.a. Superpixels: The image is segmented into disjoint components, widely over-segmenting the scene. In this scenario, a pixelwise classifier is trained on the convolutional feature vectors, and a simple vote is done for each component, to assign a single class per component. This method is simple and effective, but imposes a fixed level of segmentation, which can be suboptimal.

2.b. Conditional random field over superpixels: a conditional random field is defined over a set of superpixels. Compared to the previous, simpler method, this post-processing models joint probabilities at the level of the scene, and is useful to avoid local aberrations (*e.g.* a person in the sky). That kind of approach is widely used in the computer vision community, and we show that our learned multiscale feature representation essentially makes the use of a global random field much less useful: most scene-level relationships seem to be already captured by it.

2.c. Multilevel cut with class purity criterion: A family of segmentations is constructed over the image to analyze the scene at multiple levels. In the simplest case, this family might be a segmentation tree; in the most general case it can be any set of segmentations, for example a collection of superpixels either produced using the same algorithm with different parameter tunings or produced by different algorithms. Each segmentation component is represented by the set of feature vectors that fall into it: the component is encoded by a spatial grid of aggregated feature vectors. The aggregated feature vector of each grid cell is computed by a component-wise max pooling of the feature vectors centered on all the pixels that fall into the grid cell. This produces a scale-invariant representation of the segment and its surrounding. A classifier is then applied to the aggregated feature grid of each node. This classifier is trained to estimate the histogram of all object categories present in the component. A subset of the components is then selected such that they cover the entire image. These components are selected so as to minimize the average "impurity" of the class distribution in a procedure that we name "optimal cover". The class "impurity" is defined as the entropy of the class distribution. The choice of the cover thus attempts to find a consistent overall segmentation in which each segment contains pixels belonging to only one of the learned categories. This simple method allows us to consider full families of segmentation components, rather than a unique, predetermined segmentation (*e.g.* a single set of superpixels).

All the steps in the process have a complexity linear (or almost linear) in the number of pixels. The bulk of the computation resides in the convolutional network feature extractor. The resulting system is very fast, producing a full parse of a 320×240 image in less than a second on a conventional CPU, and in less than 100ms using dedicated

hardware, opening the door to real-time applications. Once trained, the system is parameter free, and requires no adjustment of thresholds or other knobs.

An early version of this work was first published in (34). This journal version reports more complete experiments, comparisons and higher results.

2.2.2 Multiscale feature extraction for scene parsing

The model proposed in this chapter, depicted on Figure 2.1, relies on two complementary image representations. In the first representation, an image patch is seen as a point in $\mathbb{R}^P$, and we seek to find a transform $f : \mathbb{R}^P \to \mathbb{R}^Q$ that maps each patch into $\mathbb{R}^Q$, a space where it can be classified linearly. This first representation typically suffers from two main problems when using a classical convolutional network, where the image is divided following a grid pattern: (1) the window considered rarely contains an object that is properly centered and scaled, and therefore offers a poor observation basis to predict the class of the underlying object, (2) integrating a large context involves increasing the grid size, and therefore the dimensionality P of the input; given a finite amount of training data, it is then necessary to enforce some invariance in the function f itself. This is usually achieved by using pooling/subsampling layers, which in turn degrades the ability of the model to precisely locate and delineate objects. In this chapter, f is implemented by a multiscale convolutional network, which allows integrating large contexts (as large as the complete scene) into local decisions, yet still remaining manageable in terms of parameters/dimensionality. This multiscale model, in which weights are shared across scales, allows the model to capture long-range interactions, without the penalty of extra parameters to train. This model is described in Section 2.2.2.1.

In the second representation, the image is seen as an edge-weighted graph, on which one or several over-segmentations can be constructed. The components are spatially accurate, and naturally delineate the underlying objects, as this representation conserves pixel-level precision. Section 2.2.3 describes multiple strategies to combine both representations. In particular, we describe in Section 2.2.3.3 a method for analyzing a family of segmentations (at multiple levels). It can be used as a solution to the first problem exposed above: assuming the capability of assessing the quality of all the components

in this family of segmentations, a system can automatically choose its components so as to produce the best set of predictions.

2.2.2.1 Scale-invariant, scene-level feature extraction

Good internal representations are hierarchical. In vision, pixels are assembled into edglets, edglets into motifs, motifs into parts, parts into objects, and objects into scenes. This suggests that recognition architectures for vision (and for other modalities such as audio and natural language) should have multiple trainable stages stacked on top of each other, one for each level in the feature hierarchy. Convolutional Networks (ConvNets) provide a simple framework to learn such hierarchies of features.

Convolutional Networks (64, 65) are trainable architectures composed of multiple stages. The input and output of each stage are sets of arrays called *feature maps*. For example, if the input is a color image, each feature map would be a 2D array containing a color channel of the input image (for an audio input each feature map would be a 1D array, and for a video or volumetric image, it would be a 3D array). At the output, each feature map represents a particular feature extracted at all locations on the input. Each stage is composed of three layers: a *filter bank layer*, a *non-linearity layer*, and a *feature pooling layer*. A typical ConvNet is composed of one, two or three such 3-layer stages, followed by a classification module. Because they are trainable, arbitrary input modalities can be modeled, beyond natural images.

Our feature extractor is a three-stage convolutional network. The first two stages contain a bank of filters producing multiple feature maps, a point-wise non-linear mapping and a spatial pooling followed by subsampling of each feature map. The last layer only contains a bank of filters. The filters (convolution kernels) are subject to training. Each filter is applied to the input feature maps through a 2D convolution operation, which detects local features at all locations on the input. Each filter bank of a convolutional network produces features that are equivariant under shifts, *i.e.* if the input is shifted, the output is also shifted but otherwise unchanged.

While convolutional networks have been used successfully for a number of image labeling problems, image-level tasks such as full-scene understanding (pixel-wise labeling, or any dense feature estimation) require the system to model complex interactions at the scale of complete images, not simply within a patch. To view a large contextual

window at full resolution, a convolutional network would have to be unmanageably large.

The solution is to use a multiscale approach. Our multiscale convolutional network overcomes these limitations by extending the concept of spatial weight replication to the scale space. Given an input image $\mathbf{I}$, a multiscale pyramid of images $\mathbf{X}_s$, $\forall s \in \{1, \ldots, N\}$ is constructed, where $\mathbf{X}_1$ has the size of $\mathbf{I}$. The multiscale pyramid can be a Laplacian pyramid, and is typically pre-processed, so that local neighborhoods have zero mean and unit standard deviation. Given a classical convolutional network f_s with parameters θ_s, the multiscale network is obtained by instantiating one network per scale s, and sharing all parameters across scales: $\theta_s = \theta_0$, $\forall s \in \{1, \ldots, N\}$.

We introduce the following convention: banks of images will be seen as three dimensional arrays in which the first dimension is the number of independent feature maps, or images, the second is the height of the maps and the third is the width. The output state of the L-th stage is denoted $\mathbf{H}_L$.

The maps in the pyramid are computed using a scaling/normalizing function g_s as $\mathbf{X}_s = g_s(\mathbf{I})$, for all $s \in \{1, \ldots, N\}$.

For each scale s, the convolutional network f_s can be described as a sequence of linear transforms, interspersed with non-linear symmetric squashing units (typically the tanh function (66)), and pooling/subsampling operators. For a network f_s with L layers, we have:

$$f_s(\mathbf{X}_s; \theta_s) = \mathbf{W}_L \mathbf{H}_{L-1}, \tag{2.1}$$

where the vector of hidden units at layer l is

$$\mathbf{H}_l = \text{pool}(\tanh(\mathbf{W}_l \mathbf{H}_{l-1} + \mathbf{b}_l)) \tag{2.2}$$

for all $l \in \{1, \ldots, L-1\}$, with $\mathbf{b}_l$ a vector of bias parameters, and $\mathbf{H}_0 = \mathbf{X}_s$. The matrices $\mathbf{W}_l$ are Toeplitz matrices, therefore each hidden unit vector $\mathbf{H}_l$ can be expressed as a regular convolution between kernels from $\mathbf{W}_l$ and the previous hidden unit vector $\mathbf{H}_{l-1}$, squashed through a tanh, and pooled spatially. More specifically,

$$\mathbf{H}_{lp} = \text{pool}(\tanh(b_{lp} + \sum_{q \in \text{parents}(p)} \mathbf{w}_{lpq} * \mathbf{H}_{l-1,q})). \tag{2.3}$$

The filters $\mathbf{W}_l$ and the biases $\mathbf{b}_l$ constitute the trainable parameters of our model, and are collectively denoted θ_s. The function tanh is a point-wise non-linearity, while pool is a function that considers a neighborhood of activations, and produces one activation per neighborhood. In all our experiments, we use a max-pooling operator, which takes the maximum activation within the neighborhood. Pooling over a small neighborhood provides built-in invariance to small translations.

Finally, the outputs of the N networks are upsampled and concatenated so as to produce $\mathbf{F}$, a map of feature vectors of size N times the size of $\mathbf{f}_1$, which can be seen as local patch descriptors and scene-level descriptors

$$\mathbf{F} = [\mathbf{f}_1, u(\mathbf{f}_2), \ldots, u(\mathbf{f}_N)], \tag{2.4}$$

where u is an upsampling function.

As mentioned above, weights are shared between networks f_s. Intuitively, imposing complete weight sharing across scales is a natural way of forcing the network to learn scale invariant features, and at the same time reduce the chances of over-fitting. The more scales used to jointly train the models $f_s(\theta_s)$ the better the representation becomes for all scales. Because image content is, in principle, scale invariant, using the same function to extract features at each scale is a reasonable assumption.

2.2.2.2 Learning discriminative scale-invariant features

As described in Section 2.2.2.1, feature vectors in $\mathbf{F}$ are obtained by concatenating the outputs of multiple networks f_s, each taking as input a different image in a multiscale pyramid.

Ideally a linear classifier should produce the correct categorization for all pixel locations i, from the feature vectors $\mathbf{F}_i$. We train the parameters θ_s to achieve this goal, using the multiclass *cross entropy* loss function. Let $\hat{\mathbf{c}}_i$ be the normalized prediction vector from the linear classifier for pixel i. We compute normalized predicted probability distributions over classes $\hat{\mathbf{c}}_{i,a}$ using the *softmax* function, *i.e.*

$$\hat{\mathbf{c}}_{i,a} = \frac{e^{\mathbf{w}_a^T \mathbf{F}_i}}{\sum_{b \in \text{classes}} e^{\mathbf{w}_b^T \mathbf{F}_i}}, \tag{2.5}$$

where $\mathbf{w}$ is a temporary weight matrix only used to learn the features. The cross entropy between the predicted class distribution $\hat{\mathbf{c}}$ and the target class distribution $\mathbf{c}$

penalizes their deviation and is measured by

$$L_{\text{cat}} = -\sum_{i \in \text{pixels}} \sum_{a \in \text{classes}} \mathbf{c}_{i,a} \ln(\hat{\mathbf{c}}_{i,a}). \tag{2.6}$$

The true target probability $\mathbf{c}_{i,a}$ of class a to be present at location i can either be a distribution of classes at location i, in a given neighborhood or a hard target vector: $\mathbf{c}_{i,a} = 1$ if pixel i is labeled a, and 0 otherwise. For training maximally discriminative features, we use hard target vectors in this first stage.

Once the parameters θ_s are trained, the classifier in Eq 2.5 is discarded, and the feature vectors $\mathbf{F}_i$ are used using different strategies, as described in Section 2.2.3.

2.2.3 Scene labeling strategies

The simplest strategy for labeling the scene is to use the linear classifier described in Section 2.2.2.2, and assign each pixel with the *argmax* of the prediction at its location. More specifically, for each pixel i

$$l_i = \arg\max_{a \in \text{classes}} \hat{\mathbf{c}}_{i,a}. \tag{2.7}$$

The resulting labeling l, although fairly accurate, is not satisfying visually, as it lacks spatial consistency, and precise delineation of objects. In this section, we explore three strategies to produce spatially more appealing labelings.

2.2.3.1 Superpixels

Predicting the class of each pixel independently from its neighbors yields noisy predictions. A simple cleanup can be obtained by forcing local regions of same color intensities to be assigned a single label.

As in (39, 41), we compute superpixels, following the method proposed by (35), to produce an over-segmentation of the image. We then classify each location of the image densely, and aggregate these predictions in each superpixel, by computing the average class distribution within the superpixel.

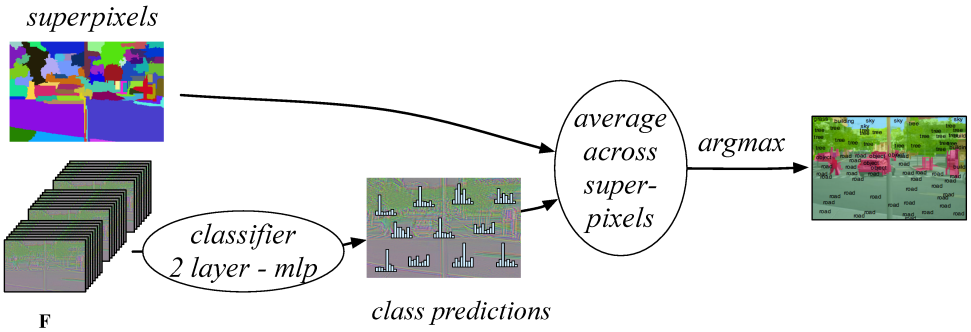

Figure 2.2: Superpixels - First labeling strategy from the features: using superpixels as described in Section 2.2.3.1.

For this method, the pixelwise distributions $\hat{\mathbf{d}}_k$ at superpixel k are predicted from the feature vectors $\mathbf{F}_i$ using a two-layer neural network:

$$\mathbf{y}_i = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{F}_i + \mathbf{b}_1), \tag{2.8}$$

$$\hat{\mathbf{d}}_{i,a} = \frac{e^{\mathbf{y}_{i,a}}}{\sum_{b \in \text{classes}} e^{\mathbf{y}_{i,b}}}, \tag{2.9}$$

$$L_{\text{cat}} = -\sum_{i \in \text{pixels}} \sum_{a \in \text{classes}} \mathbf{d}_{i,a} \ln(\hat{\mathbf{d}}_{i,a}), \tag{2.10}$$

$$\hat{\mathbf{d}}_{k,a} = \frac{1}{\text{s}(k)} \sum_{i \in k} \hat{\mathbf{d}}_{i,a}, \tag{2.11}$$

with $\mathbf{d}_i$ the groundtruth distribution at location i, and $\text{s}(k)$ the surface of component k. Matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ are the trainable parameters of the classifier. Using a two-layer neural network, as opposed to the simple linear classifier used in Section 2.2.2.2, allows the system to capture non-linear relationships between the features at different scales. In this case, the final labeling for each component k is given by

$$l_k = \arg\max_{a \in \text{classes}} \hat{\mathbf{d}}_{k,a}. \tag{2.12}$$

The pipeline is depicted in Figure 2.2.

2.2.3.2 Conditional Random Fields

The local assignment obtained using superpixels does not involve a global understanding of the scene. In this section, we implement a classical CRF model, constructed on

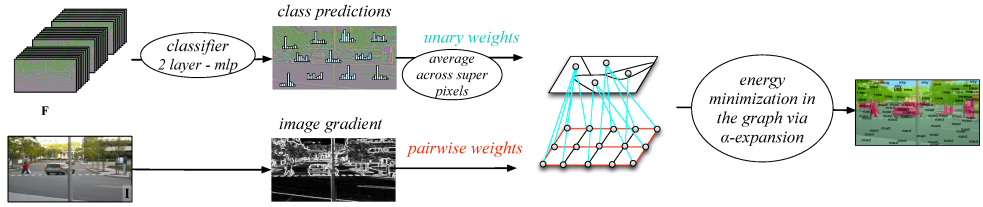

Figure 2.3: CRFs - Second labeling strategy from the features: using a CRF, described in Section 2.2.3.2.

the superpixels. This is a quite standard approach for image labeling. Our multi-scale convolutional network already has the capability of modeling global relationships within a scene, but might still be prone to errors, and can benefit from a CRF, to impose consistency and coherency between labels, at test time.

A common strategy for labeling a scene consists in associating the image to a graph and define an energy function whose optimal solution corresponds to the desired segmentation (39, 94).

For this purpose, we define a graph $G = (V, E)$ with vertices $v \in V$ and edges $e \in E \subseteq V \times V$. Each pixel in the image is associated to a vertex, and edges are added between every neighboring nodes. An edge, e, spanning two vertices, v_i and v_j , is denoted by e_{ij}. The Conditional Random Field (CRF) energy function is typically composed of a unary term enforcing the variable l to take values close to the predictions $\hat{\mathbf{d}}$ and a pairwise term enforcing regularity or local consistency of l. The CRF energy to minimize is given by

$$E(l) = \sum_{i \in V} \Phi(\hat{\mathbf{d}}_i, l_i) + \gamma \sum_{e_{ij} \in E} \Psi(l_i, l_j). \tag{2.13}$$

We considered as unary terms

$$\Phi(\hat{\mathbf{d}}_{i,a}, l_i) = \exp\left(-\alpha \hat{\mathbf{d}}_{i,a}\right) \mathbf{1}(l_i \neq a), \tag{2.14}$$

where $\hat{\mathbf{d}}_{i,a}$ corresponds to the probability of class a to be present at a pixel i computed as in Section 2.2.3.1, and $\mathbf{1}(\cdot)$ is an indicator function that equals one if the input is true, and zero otherwise.

The pairwise term consists of

$$\Psi(l_i, l_j) = \exp\left(-\beta \|\nabla I\|_i\right) \mathbf{1}(l_i \neq l_j) \tag{2.15}$$

where $\|\nabla I\|_i$ is the ℓ_2 norm of the gradient of the image I at a pixel i. Details on the parameters used are given in the experimental section.

The CRF energy (2.13) is minimized using alpha-expansions (15, 16). An illustration of the procedure appears in Figure 2.3.

2.2.3.3 Parameter-free multilevel parsing

One problem subsists with the two methods presented above: the observation level problem. An object, or object part, can be easily classified once it is segmented at the right level. The two methods above are based on an arbitrary segmentation of the image, which typically decomposes it into segments that are too small, or, more rarely, too large.

In this section, we propose a method to analyze a family of segmentations and automatically discover the best observation level for each pixel in the image. One special case of such families is the segmentation tree, in which components are hierarchically organized. Our method is not restricted to such trees, and can be used for arbitrary sets of neighborhoods.

In Section 2.2.3.3 we formulate the search for the most adapted neighborhood of a pixel as an optimization problem. The construction of the cost function that is minimized is then described in Section 2.2.3.3.

Optimal purity cover We define the neighborhood of a pixel as a connected component that contains this pixel. Let C_k, $\forall k \in \{1, \ldots, K\}$ be the set of all possible connected components of the lattice defined on image I, and let S_k be a cost associated to each of these components. For each pixel i, we wish to find the index $k^*(i)$ of the component that best explains this pixel, that is, the component with the minimal cost $S_{k^*(i)}$:

$$k^*(i) = \underset{k \mid i \in C_k}{\operatorname{argmin}} S_k. \tag{2.16}$$

Note that components $C_{k^*(i)}$ form a non-disjoint set that covers the lattice. The set is always guaranteed to form a cover because of its definition: each pixel is assigned a component with minimal cost, therefore these components cover the entire image. Note also that the overall cost $S^* = \sum_i S_{k^*(i)}$ is minimal.

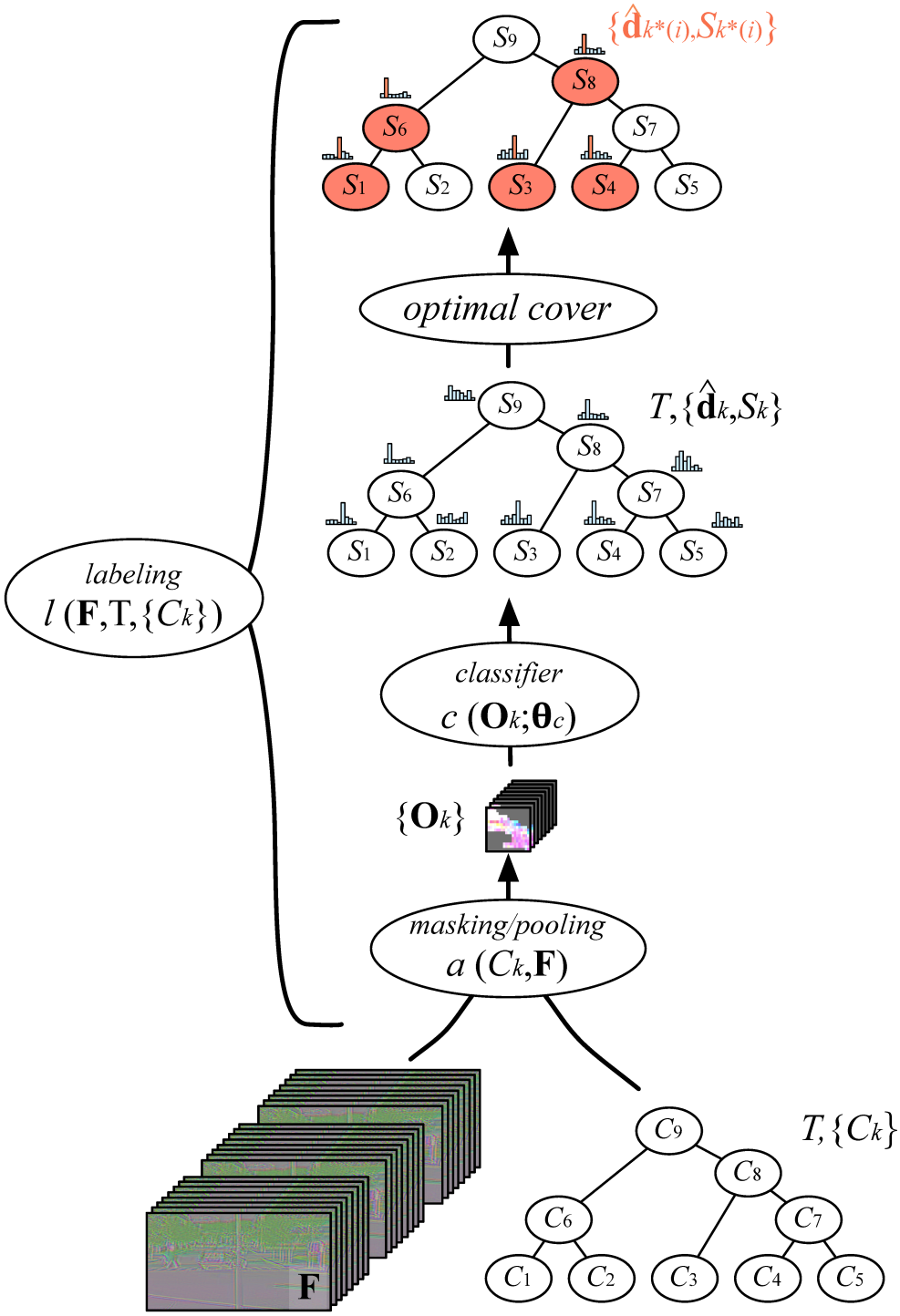

Figure 2.4: Optimal Cover - Third labeling strategy from the features: using a family of of segmentations, as described in Section 2.2.3.3. In this figure, the family of segmentations is a segmentation tree. The segment associated with each node in the tree is encoded by a spatial grid of feature vectors pooled in the segment's region. A classifier is then applied to all the aggregated feature grids to produce a histogram of categories, the entropy of which measures the "impurity" of the segment. Each pixel is then labeled by the minimally-impure node above it, which is the segment that best "explains" the pixel.

In practice, the set of components C_k is too large, and only a subset of it can be considered. A classical technique to reduce the set of components is to consider a hierarchy of segmentations (6, 78), that can be represented as a tree T. This was previously explored in (34). Solving Eq 2.16 on T consists of the following procedure: for each pixel (leaf) i, the optimal component $C_{k^*(i)}$ is the one along the path between the leaf and the root with minimal cost $S_{k^*(i)}$. The optimal cover is the union of all these components. For efficiency purposes, it can be done simply by exploring the tree in a depth-first search manner, and finding the component with minimal weight along each branch. The complexity of the optimal cover procedure is then linear in the number of components in the tree. Figure 2.5 illustrates the procedure.

Another technique to reduce the set of components considered is to compute a set of segmentations using different merging thresholds. In Section 2.2.4, we use such an approach, by computing multiple levels of the Felzenszwalb algorithm (35). The Felzenszwalb algorithm is not strictly monotonic, so the structure obtained cannot be cast into a tree: rather, it has a general graph form, in which each pixel belongs to as many superpixels as levels explored. Solving Eq 2.16 in this case consists of the following procedure: for each pixel i, the optimal component $C_{k^*(i)}$ is the one among all the segmentations with minimal cost $S_{k^*(i)}$. Thus the complexity to produce a cover on the family of components is linear on the number of pixels, but with a constant that is proportional to the number of levels explored.

Producing the confidence costs Given a set of components C_k, we explain how to produce all the confidence costs S_k. These costs represent the class purity of the associated components. Given the groundtruth segmentation, we can compute the cost as being the entropy of the distribution of classes present in the component. At test time, when no groundtruth is available, we need to define a function that can predict this cost by simply looking at the component. We now describe a way of achieving this, as illustrated in Figure 2.6.

Given the scale-invariant features $\mathbf{F}$, we define a compact representation to describe objects as an elastic spatial arrangement of such features. In other terms, an object, or category in general, can be best described as a spatial arrangement of features, or parts. We define a simple attention function a used to mask the feature vector map with each component C_k, producing a set of K masked feature vector patterns

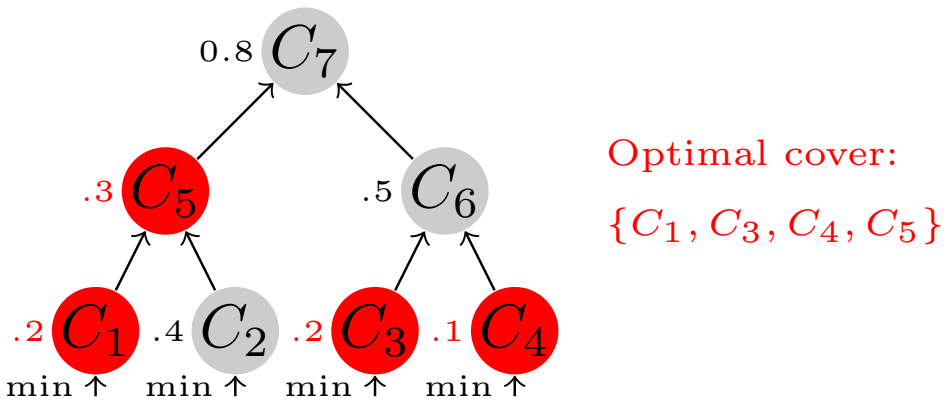

Figure 2.5: Optimal Cover on a Tree: - Finding the optimal cover on a tree. The numbers next to the components correspond to the entropy scores S_i. For each pixel (leaf) i, the optimal component $C_{k^*(i)}$ is the one along the path between the leaf and the root with minimal cost $S_{k^*(i)}$. The optimal cover is the union of all these components. In this example, the optimal cover $\{C_1, C_3, C_4, C_5\}$ will result in a segmentation in disjoint sets $\{C_1, C_2, C_3, C_4\}$, with the subtle difference that component C_2 will be labelled with the class of C_5, as C_5 is the best observation level for C_2. The generalization to a family of segmentations is straightforward (see text).

$\{\mathbf{F} \bigcap C_k\}$, $\forall k \in \{1, \ldots, K\}$. The function a is called an attention function because it suppresses the background around the component being analyzed. The patterns $\{\mathbf{F} \bigcap C_k\}$ are resampled to produce fixed-size representations. In our model the sampling is done using an elastic max-pooling function, which remaps input patterns of arbitrary size into a fixed $G \times G$ grid. This grid can be seen as a highly invariant representation that encodes spatial relations between an object's attributes/parts. This representation is denoted $\mathbf{O}_k$. Some nice properties of this encoding are: (1) elongated, or in general ill-shaped objects, are nicely handled, (2) the dominant features are used to represent the object, combined with background subtraction, the features pooled represent solid basis functions to recognize the underlying object.

Once we have the set of object descriptors $\mathbf{O}_k$, we define a function $c : \mathbf{O}_k \rightarrow [0, 1]^{N_c}$ (where N_c is the number of classes) as predicting the distribution of classes present in component C_k. We associate a cost S_k to this distribution. In this chapter, c is implemented as a simple 2-layer neural network, and S_k is the entropy of the predicted distribution. More formally, let $\mathbf{O}_k$ be the feature vector associated with component C_k, $\hat{\mathbf{d}}_k$ the predicted class distribution, and S_k the cost associated to this distribution.

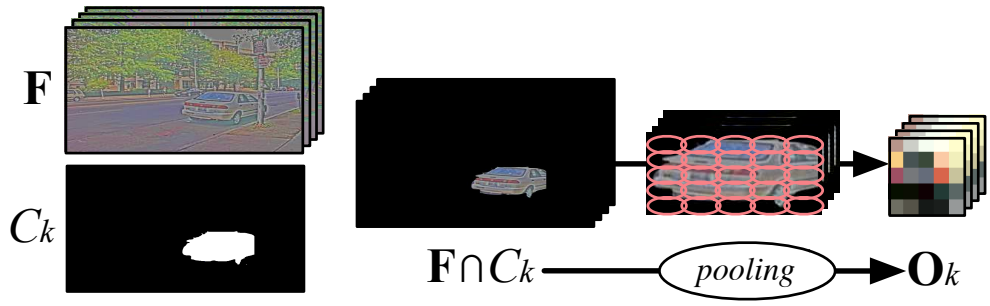

Figure 2.6: Max Sampling - The shape-invariant attention function a. For each component C_k in the family of segmentations T, the corresponding image segment is encoded by a spatial grid of feature vectors that fall into this segment. The aggregated feature vector of each grid cell is computed by a component-wise max pooling of the feature vectors centered on all the pixels that fall into the grid cell; this produces a scale-invariant representation of the segment and its surroundings. The result, $\mathbf{O}_k$, is a descriptor that encodes spatial relations between the underlying object's parts. The grid size was set to 3×3 for all our experiments.

We have

$$\mathbf{y}_k = \mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{O}_k + \mathbf{b}_1), \tag{2.17}$$

$$\hat{\mathbf{d}}_{k,a} = \frac{e^{\mathbf{y}_{k,a}}}{\sum_{b \in \text{classes}} e^{\mathbf{y}_{k,b}}}, \tag{2.18}$$

$$S_k = -\sum_{a \in \text{classes}} \mathbf{d}_{k,a} \ln(\hat{\mathbf{d}}_{k,a}), \tag{2.19}$$

with $\mathbf{d}_k$ the groundtruth distribution for component k. Matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ are noted θ_c, and represent the trainable parameters of c. These parameters need to be learned over the complete set of segmentation families, computed on the entire training set available. The training procedure is described in Section 2.2.3.3.

For each component C_k chosen by the optimal purity cover (Section 2.2.3.3) the label is produced by:

$$l_k = \arg\max_{a \in \text{classes}} \hat{\mathbf{d}}_{k,a} \quad C_k \in \text{cut}. \tag{2.20}$$

Training procedure Let $\mathcal{F}$ be the set of all feature maps in the training set, and $\mathcal{T}$ the set of all families of segmentations. We construct the segmentation collections $(T)_{T \in \mathcal{T}}$ on the entire training set, and, for all $T \in \mathcal{T}$ train the classifier c to predict the distribution of classes in component $C_k \in T$, as well as the costs S_k.

Given the trained parameters θ_s, we build $\mathcal{F}$ and $\mathcal{T}$, *i.e.* we compute all vector maps $\mathbf{F}$ and segmentation collections T on all the training data available, so as to produce a new training set of descriptors $\mathbf{O}_k$. This time, the parameters θ_c of the classifier c are trained to minimize the KL-divergence between the true (known) distributions of labels $\mathbf{d}_k$ in each component, and the prediction from the classifier $\hat{\mathbf{d}}_k$ (Eq 2.18):

$$l_{div} = \sum_{a \in \text{classes}} \hat{\mathbf{d}}_{k,a} \ln(\frac{\hat{\mathbf{d}}_{k,a}}{\mathbf{d}_{k,a}}). \tag{2.21}$$

In this setting, the groundtruth distributions $\mathbf{d}_k$ are not hard target vectors, but normalized histograms of the labels present in component C_k. Once the parameters θ_c are trained, $\hat{\mathbf{d}}_k$ accurately predicts the distribution of labels, and Eq 2.19 is used to assign a purity cost to the component.

2.2.4 Experiments

We report our semantic scene understanding results on three different datasets: "Stanford Background" on which related state-of-the-art methods report classification errors, and two more challenging datasets with a larger number of classes: "SIFT Flow" and "Barcelona". The Stanford Background dataset (42) contains 715 images of outdoor scenes composed of 8 classes, chosen from other existing public datasets so that all the images are outdoor scenes, have approximately 320×240 pixels, where each image contains at least one foreground object. We use the evaluation procedure introduced in (42), 5-fold cross validation: 572 images used for training, and 143 for testing. The SIFT Flow dataset (72) is composed of $2,688$ images, that have been thoroughly labeled by LabelMe users, and split in $2,488$ training images and 200 test images. The authors used synonym correction to obtain 33 semantic labels. The Barcelona dataset, as described in (98), is derived from the LabelMe subset used in (90). It has $14,871$ training and 279 test images. The test set consists of street scenes from Barcelona, while the training set ranges in scene types but has no street scenes from Barcelona. Synonyms were manually consolidated by (98) to produce 170 unique labels.

To evaluate the representation from our multiscale convolutional network, we report results from several experiments on the Stanford Background dataset: (1) a system based on a plain convolutional network alone; (2) the multiscale convolutional network

	Pixel Acc.	Class Acc.	CT (sec.)
Gould *et al.* 2009 (42)	76.4%	-	10 to 600s
Munoz *et al.* 2010 (76)	76.9%	66.2%	12s
Tighe *et al.* 2010 (98)	77.5%	-	10 to 300s
Socher *et al.* 2011 (97)	78.1%	-	?
Kumar *et al.* 2010 (57)	79.4%	-	$< 600s$
Lempitzky *et al.* 2011 (70)	**81.9%**	72.4%	$> 60s$
singlescale convnet	66.0 %	56.5 %	0.35s
multiscale convnet	78.8 %	72.4%	0.6s
multiscale net + superpixels	80.4%	74.56%	**0.7s**
multiscale net + gPb + cover	80.4%	75.24%	61s
multiscale net + CRF on gPb	81.4%	**76.0**%	60.5s

Table 2.1: Performance of our system on the Stanford Background dataset (42): per-pixel / average per-class accuracy. The third column reports compute times, as reported by the authors. Our algorithms were computed using a 4-core Intel i7.

	Pixel Acc.	Class Acc.
Liu *et al.* 2009 (72)	74.75%	-
Tighe *et al.* 2010 (98)	76.9%	29.4%
raw multiscale net[1]	67.9%	45.9%
multiscale net + superpixels[1]	71.9%	**50.8%**
multiscale net + cover[1]	72.3%	**50.8%**
multiscale net + cover[2]	**78.5%**	29.6%

Table 2.2: Performance of our system on the SIFT Flow dataset (72): per-pixel / average per-class accuracy. Our multiscale network is trained using two sampling methods: [1]balanced frequencies, [2]natural frequencies. We compare the results of our multiscale network with the raw (pixelwise) classifier, Felzenszwalb superpixels (35) (one level), and our optimal cover applied to a stack of 10 levels of Felzenszwalb superpixels. Note: the threshold for the single level was picked to yield the best results; the cover automatically finds the best combination of superpixels.

presented in Section 2.2.2.1, with raw pixelwise prediction; (3) superpixel-based predictions, as presented in Section 2.2.3.1; (4) CRF-based predictions, as presented in Section 2.2.3.2; (5) cover-based predictions, as presented in Section 2.2.3.3.

Results are reported in Table 2.1, and compared with related works. Our model achieves very good results in comparison with previous approaches. Methods of (57, 70) achieve similar or better performances on this particular dataset but to the price of several minutes to parse one image.

We then demonstrate that our system scales nicely when augmenting the number of classes on two other datasets, in Tables 2.2 and 2.3. Results on these datasets were obtained using our cover-based method, from Section 2.2.3.3. Example parses on the SIFT Flow dataset are shown on Figure 2.9.

For the SIFT Flow and Barcelona datasets, we experimented with two sampling methods when learning the multiscale features: respecting natural frequencies of classes, and balancing them so that an equal amount of each class is shown to the network. Balancing class occurrences is essential to model the conditional likelihood of each class (*i.e.* ignore their prior distribution). Both results are reported in Table 2.2. Training with balanced frequencies allows better discrimination of small objects, and although it decreases the overall pixelwise accuracy, it is more correct from a recognition point of view. Frequency balancing is used on the Stanford Background dataset, as it consistently gives better results. For the Barcelona dataset, both sampling methods are used as well, but frequency balancing worked rather poorly in that case. This can be explained by the fact that this dataset has a large amount of classes with very few training examples. These classes are therefore extremely hard to model, and overfitting occurs much faster than for the SIFT Flow dataset. Results are shown on Table 2.3.

Results in Table 2.1 demonstrate the impressive computational advantage of convolutional networks over competing algorithms. Exploiting the parallel structure of this special network, by computing convolutions in parallel, allows us to parse an image of size 320×240 in less than one second on a 4-core Intel i7 laptop. Using GPUs or other types of dedicated hardware, our scene parsing model can be run in real-time (*i.e.* at more than 10fps).

	Pixel Acc.	Class Acc.
Tighe *et al.* 2010 (98)	66.9%	7.6%
raw multiscale net[1]	37.8%	**12.1**%
multiscale net + superpixels[1]	44.1%	**12.4**%
multiscale net + cover[1]	46.4%	**12.5**%
multiscale net + cover[2]	**67.8**%	**9.5**%

Table 2.3: Performance of our system on the Barcelona dataset (98): per-pixel / average per-class accuracy. Our multiscale network is trained using two sampling methods: [1]balanced frequencies, [2]natural frequencies. We compare the results of our multiscale network with the raw (pixelwise) classifier, Felzenszwalb superpixels (35) (one level), and our optimal cover applied to a stack of 10 levels of Felzenszwalb superpixels. Note: the threshold for the single level was picked to yield the best results; the cover automatically finds the best combination of superpixels.

2.2.4.1 Multiscale feature extraction

For all experiments, we use a 3-stage convolutional network. The first two layers of the network are composed of a bank of filters of size 7×7 followed by tanh units and 2×2 max-pooling operations. The last layer is a simple filter bank. The filters and pooling dimensions were chosen by a grid search. The input image is transformed into YUV space, and a Laplacian pyramid is constructed from it. The Y, U and V channels of each scale in the pyramid are then independently locally normalized, such that each local 15×15 patch has zero-mean and unit variance. For these experiments, the pyramid consists of 3 rescaled versions of the input ($N = 3$), in octaves: $320 \times 240,\ 160 \times 120,\ 80 \times 60$.

The network is then applied to each 3-dimension input map $\mathbf{X}_s$. This input is transformed into a 16-dimension feature map, using a bank of 16 filters (10 connected to the Y channel, 6 connected to the U and V channels). The second layer transforms this 16-dimension feature map into a 64-dimension feature map, each map being produced by a combination of 8 randomly selected feature maps from the previous layer. Finally the 64-dimension feature map is transformed into a 256-dimension feature map, each map being produced by a combination of 32 randomly selected feature maps from the previous layer.

The outputs of each of the 3 networks are then upsampled and concatenated, so as to produce a $256 \times 3 = 768$-dimension feature vector map $\mathbf{F}$. Given the filter

sizes, the network has a field of view of 46×46, at each scale, which means that a feature vector in $\mathbf{F}$ is influenced by a 46×46 neighborhood at full resolution, a 92×92 neighborhood at half resolution, and a 184×184 neighborhood at quarter resolution. These neighborhoods are shown in Figure 2.1.

The network is trained on all 3 scales in parallel, using stochastic gradient descent with no second-order information, and mini-batches of size 1. Simple grid-search was performed to find the best learning rate (10^{-3}) and regularization parameters (L2 coefficient: 10^{-5}), using a holdout of 10% of the training data for validation. The holdout is also used to select the best network, *i.e.* the network that generalizes the most on the holdout.

Convergence, that is, maximum generalization performance, is typically attained after between 10 to 50 million patches have been seen during stochastic gradient descent. This typically represents between two to five days of training. No special hardware (GPUs) was used for training.

The convolutional network has roughly 0.5 million trainable parameters. To ensure that features do not overfit some irrelevant biases present in the data, jitter – horizontal flipping of all images, rotations between -8 and 8 degrees, and rescaling between 90 and 110% – was used to artificially expand the size of the training data. These additional distortions are applied during training, before loading a new training point, and are sampled from uniform distributions. Jitter was shown to be crucial for low-level feature learning in the works of (95) and (21).

For our baseline, we trained a single-scale network and a three-scale network as raw site predictors, for each location i, using the classification loss L_{cat} defined in Eq 2.10, with the two-layer neural network defined in Eq 2.9. Table 2.1 shows the clear advantage of the multi-scale representation, which captures scene-level dependencies, and can classify more pixels accurately. Without an explicit segmentation model, the visual aspect of the predictions still suffers from inaccurate object delineation.

2.2.4.2 Parsing with superpixels

The results obtained with the strategy presented in section 2.2.3.1 demonstrate the quality of our multiscale features, by reaching a very high classification accuracy on all three datasets. This simple strategy is also a real fit for real time applications, taking

only an additional 0.2 second to label a 320×240 image on Intel i7 CPU. An example of result is given in Figure 2.7.

The 2−layer neural network used for this method (Eq 2.9) has 768 input units, 1024 hidden units; and as many output units as classes in each dataset. This neural network is trained with no regularization.

2.2.4.3 Multilevel parsing

Although the simple strategy of the previous section seems appealing, the results can be further improved using the multilevel approach of Section 2.2.3.3.

The family of segmentations used to find the optimal cover could be a simple segmentation tree constructed on the raw image gradient. For the Stanford Background dataset experiments, we used a more sophisticated tree based on a semantic image gradient. We used the gPb hierarchies of Arbelaez *et al.* , which are computed using spectral clustering to produce semantically consistent contours of objects. Their computation requires one minute per image.

For the SIFT Flow and Barcelona datasets, we used a cheaper technique, which does not rely on a tree: we ran the superpixel method proposed by Felzenszwalb in (35) at 10 different levels. The Felzenszwalb algorithm is not strictly monotonic, so the structure obtained cannot be cast into a tree: rather, it has a general graph form, in which each pixel belongs to 10 different superpixels. Our optimal cover algorithm can be readily applied to arbitrary structures of this type. The 10 levels were chosen such that they are linearly distributed and span a large range.

Classically, segmentation methods find a partition of the segments rather than a cover. Partitioning the segments consists in finding an optimal cut in a tree (so that each terminal node in the pruned tree corresponds to a segment). We experimented with graph-cuts to do so (14, 36), but the results were less accurate than with our optimal cover method (Stanford Background dataset only).

The 2−layer neural network c from Eq 2.17 has $3 \times 3 \times 768 = 6912$ input units (using a 3×3 grid of feature vectors from $\mathbf{F}$), 1024 hidden units; and as many output units as classes in each dataset. This rather large neural network is trained with L2 regularization (coefficient: 10^{-2}), to minimize overfitting.

Results are better than the superpixel method, in particular, better delineation is achieved (see Fig. 2.7).

2.2.4.4 Conditional random field

We demonstrate the state-of-the-art quality of our features by employing a CRF on the superpixels given by thresholding the gPb hierarchy, on the Stanford Background dataset. A similar test is performed in Lempitsky *et al.* (70), where the authors also use a CRF on the same superpixels (at the threshold 20 in the gPb hierarchy), but employ different features. Histograms of densely sampled SIFT words, colors, locations, and contour shape descriptors. They report a ratio of correctly classified pixels of 81.1% on the Stanford Background dataset. We recall that this accuracy is the best one has achieved at the present day on this dataset with a flat CRF.

In our CRF energy, we performed a grid search to set the parameters of (2.13) ($\beta = 20$, $\alpha = 0.1$ $\gamma = 200$), and used a grey level gradient. The accuracy of the resulting system is 81.4, as reported in Table 2.1. Our features are thus outperforming the best publicly available combination of handcrafted features.

2.2.4.5 Some comments on the learned features

With recent advances in unsupervised (deep) learning, learned features have become easier to analyze and understand. In this work, the entire stack of features is learned in a purely supervised manner, and yet we found that the features obtained are rather meaningful. We believe that the reason for this is the type of loss function we use, which enforces a large invariance: the system is forced to produce an invariant representation for all the locations of a given object. This type of invariance is very similar to what can be achieved using semi-supervised techniques such as Dr-LIM (44), where the loss enforces pairs of similar patches to yield a same encoding. Figure 2.10 shows an example of the features learned on the SIFT Flow dataset, for the first layer.

2.2.4.6 Some comments on real-world generalization

Now that we have compared and discussed several strategies for scene parsing based on our multiscale features, we consider taking our system in the real-world, to evaluate its generalization properties. The work of (99), measuring dataset bias, raises the question of the *generalization* of a recognition system learned on specific, publicly available datasets.

We used our multiscale features combined with classification using superpixels as described in Section 2.2.3.1, trained on the SiftFlow dataset (2,688 images, most of them taken in non-urban environments, see Table 2.2 and Figure 2.9). We collected a 360 degree movie in our workplace environment, including a street and a park, introducing difficulties such as lighting conditions and image distortions: see Figure 2.11.

The movie was built from four videos that were stitched to form a 360 degree video stream of 1280×256 images, thus creating artifacts not seen during training. We processed each frame independently, without using any temporal consistency or smoothing.

Despite all these constraints, and the rather small size of the training dataset, we observe rather convincing generalization of our models on these previously unseen scenes. The two video sequences are available at `http://www.clement.farabet.net/research.html#parsing`. Two snapshots are included in Figure 2.11. Our scene parsing system constitutes at the best of our knowledge the first approach achieving real time performance, one frame being processed in less than a second on a 4-core Intel i7. Feature extraction, which represent around 500ms on the i7 can be reduced to 60ms using dedicated FPGA hardware (32, 33).

2.2.5 Discussion and Conclusions

The main lessons from the experiments presented in this chapter are as follows:

- Using a high-capacity feature-learning system fed with raw pixels yields excellent results, when compared with systems that use engineered features. The accuracy is similar or better than competing systems, even when the segmentation hypothesis generation and the post-processing module are absent or very simple.

- Feeding the system with a wide contextual window is critical to the quality of the results. The numbers in table 2.1 show a dramatic improvement of the performance of the multi-scale convolutional network over the single scale version.

- When a wide context is taken into account to produce each pixel label, the role of the post-processing is greatly reduced. In fact, a simple majority vote of the categories within a superpixel yields state-of-the-art accuracy. This seems to suggest that contextual information can be taken into account by a feed-forward

trainable system with a wide contextual window, perhaps as well as an inference mechanism that propagates label constraints over a graphical model, but with a considerably lower computational cost.

- The use of highly sophisticated post-processing schemes, which seem so crucial to the success of other models, does not seem to improve the results significantly over simple schemes. This seems to suggest that the performance is limited by the quality of the labeling, or the quality of the segmentation hypotheses, rather than by the quality of the contextual consistency system or the inference algorithm.

- Relying heavily on a highly-accurate feed-forward pixel labeling system, while simplifying the post-processing module to its bare minimum cuts down the inference times considerably. The resulting system is dramatically faster than those that rely heavily on graphical model inference. Moreover, the bulk of the computation takes place in the convolutional network. This computation is algorithmically simple, easily parallelizable. Implementations on multi-core machines, general-purpose GPUs, Digital Signal Processors, or specialized architectures implemented on FPGAs is straightforward. This is demonstrated by the FPGA implementation (32, 33) of the feature extraction scheme presented in this chapter that runs in 60ms for an image resolution of 320×240.

This chapter demonstrates that a feed-forward convolutional network, trained end-to-end in a supervised manner, and fed with raw pixels from large patches over multiple scales, can produce state of the art performance on standard scene parsing datasets. The model does not rely on engineered features, and uses purely supervised training from fully-labeled images to learn appropriate low-level and mid-level features.

Perhaps the most surprising result is that even in the absence of any post-processing, by simply labeling each pixel with the highest-scoring category produced by the convolutional net for that location, the system yields near state-of-the-art pixel-wise accuracy, and better per-class accuracy than all previously-published results. Feeding the features of the convolutional net to various sophisticated schemes that generate segmentation hypotheses, and that find consistent segmentations and labeling by taking local constraints into account improves the results slightly, but not considerably.

While the results on datasets with few categories are good, the accuracy of the best existing scene parsing systems, including ours, is still quite low when the number of

categories is large. The problem of scene parsing is far from being solved. While the system presented here has a number of advantages and shortcomings, the framing of the scene parsing task itself is in need of refinement.

First of all, the pixel-wise accuracy is a somewhat inaccurate measure of the visual and practical quality of the result. Spotting rare objects is often more important than accurately labeling every boundary pixel of the sky (which are often in greater number). The average per-class accuracy is a step in the right direction, but not the ultimate solution: one would prefer a system that correctly spots every object or region, while giving an approximate boundary to a system that produces accurate boundaries for large regions (sky, road, grass), but fail to spot small objects. A reflection is needed on the best ways to measure the accuracy of scene labeling systems.

Scene parsing datasets also need better labels. One could imagine using scene parsing datasets with hierarchical labels, so that a window within a building would be labeled as "building" and "window". Using this kind of labeling in conjunction with graph structures on sets of labels that contain `is-part-of` relationships would likely produce more consistent interpretations of the whole scene.

The framework presented in this chapter trains the convolutional net as a pixel labeling system in isolation from the post-processing module that ensures the consistency of the labeling and its proper registration with the image regions. This requires that the convolutional net be trained with images that are fully labeled at the pixel level. One would hope that jointly fine-tuning the convolutional net and the post-processor produces better overall interpretations. Gradients can be back-propagated through the post-processor to the convolutional nets. This is reminiscent of the Graph Transformer Network model, a kind of non-linear CRF in which an un-normalized graphical model based post-processing module was trained jointly with a convolutional network for handwriting recognition (65). Unfortunately, preliminary experiments with such joint training yielded lower test-set accuracies due to overtraining.

A more important advantage of joint training would allow the use of weakly-labeled images in which only a list of objects present in the image would be given, perhaps tagged with approximate positions. This would be similar in spirit to sentence-level discriminative training methods used in speech recognition and handwriting recognition (65).

Another possible direction for improvement includes the use of objective functions that directly operates of the edge costs of neighborhood graphs in such as way that graph-cut segmentation and similar methods produce the best answer. One such objective function is Turaga's Maximin Learning (100), which pushes up the lowest edge cost along the shortest path between two points in different segments, and pushes down the highest edge cost along a path between two points in the same segment.

Our system so far has been trained using purely supervised learning applied to a fairly classical convolutional network architecture. However, a number of recent works have shown the advantage of architectural elements such as rectifying non-linearities and local contrast normalization (52). More importantly, several works have shown the advantage of using unsupervised pre-training to prime the convolutional net into a good starting point before supervised refinement (53, 54, 55, 69, 89). These methods improve the performance in the low training set size regime, and would probably improve the performance of the present system.

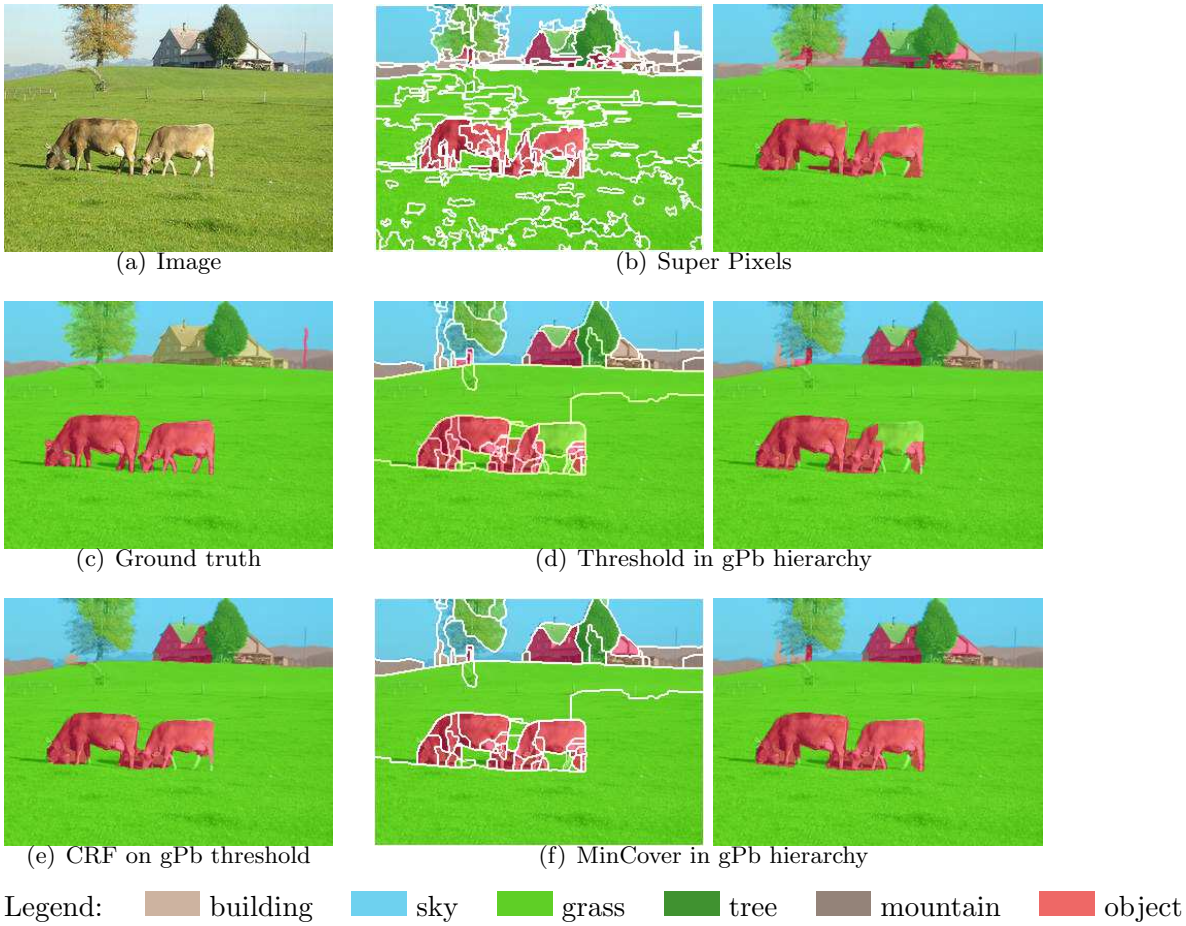

Figure 2.7: Scene Parsing Results - Example of results on the Stanford background dataset. (b),(d) and (f) show results with different labeling strategies, overlayed with superpixels (cf Section 2.2.3.1), segments results of a threshold in the gPb hierarchy (6), and segments recovered by the maximum purity approach with an optimal cover (cf 2.2.3.3). The result (c) is obtained with a CRF on the superpixels shown in (d), as described in Section 2.2.3.2.

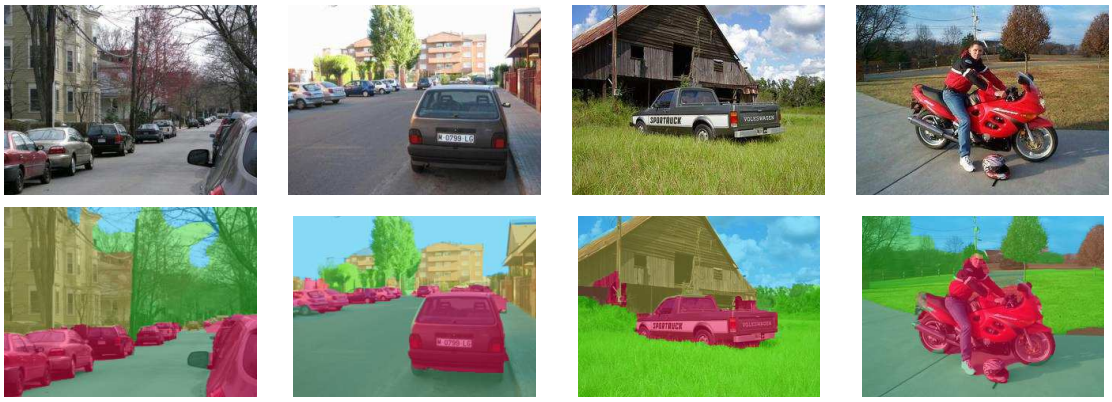

Figure 2.8: More Scene Parsing Results - More results using our multiscale convolutional network and a flat CRF on the Stanford Background Dataset.

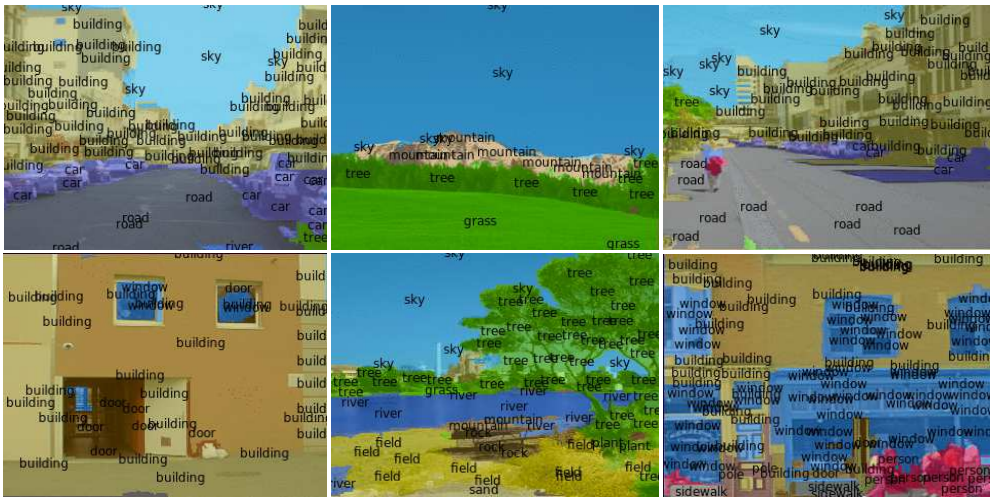

Figure 2.9: Scene Parsing Results with More Classes - Typical results achieved on the SIFT Flow dataset.

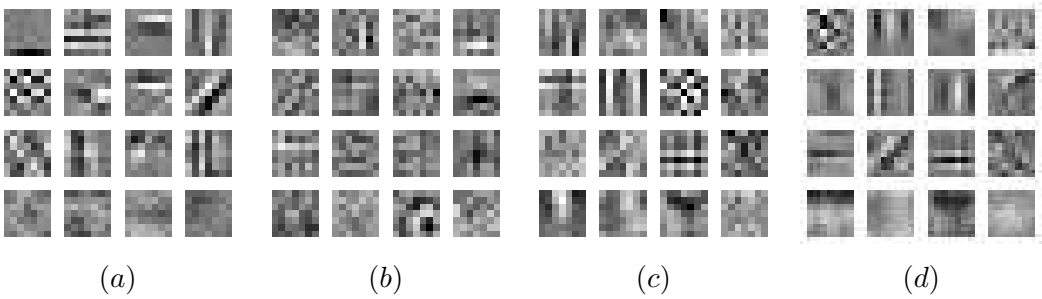

Figure 2.10: Learned Filters - Typical first layer features, learned on the SIFT Flow dataset. (a) to (c) show the 16 filters learned at each scale, when no weight sharing is used (networks at each scale are independent). (d) show the 16 filters obtained when sharing weights across all 3 scales. All the filters are 7×7. We observe typical oriented edges, and high-frequency filters. Filters at higher layers are more difficult to analyze.

Figure 2.11: Real-time scene parsing in natural conditions - training on SiftFlow dataset. We display one label per component in the final prediction.

3

A Hardware Platform for Real-time Image Understanding

3.1 Introduction

Micro-robots, unmanned aerial vehicles (UAVs), imaging sensor networks, wireless phones, and other embedded vision systems all require low cost and high-speed implementations of synthetic vision systems capable of recognizing and categorizing objects in a scene.

Many successful object recognition systems use dense features extracted on regularly-spaced patches over the input image. The majority of the feature extraction systems have a common structure composed of a filter bank (generally based on oriented edge detectors or 2D gabor functions), a non-linear operation (quantization, winner-take-all, sparsification, normalization, and/or point-wise saturation) and finally a pooling operation (max, average or histogramming). For example, the scale-invariant feature transform (SIFT (73)) operator applies oriented edge filters to a small patch and determines the dominant orientation through a winner-take-all operation. Finally, the resulting sparse vectors are added (pooled) over a larger patch to form local orientation histogram. Some recognition systems use a single stage of feature extractors (8, 28, 60, 87).

Other models like HMAX-type models (77, 93) and convolutional networks use two more layers of successive feature extractors. Different training algorithms have been used for learning the parameters of convolutional networks. In (65) and (49), pure supervised learning is used to update the parameters. However, recent works have

focused on training with an auxiliary task (4) or using unsupervised objectives (52, 54, 69, 89).

This chapter presents a scalable hardware architecture for large-scale multi-layered synthetic vision systems based on large parallel filter banks, such as convolutional networks. This hardware can also be used to accelerate the execution (and partial learning) of recent vision algorithms like SIFT and HMAX (60, 93). This system is a data-flow vision engine that can perform real-time detection, recognition and localization in mega-pixel images processed as pipelined streams. The system was designed with the goal of providing categorization of an arbitrary number of objects, while consuming very little power.

Graphics Processing Units (GPUs) are becoming a common alternative to custom hardware in vision applications, as demonstrated in (22). Their advantage over custom hardware are numerous: they are inexpensive, available in most recent computers, and easily programmable with standard development kits, such as nVidia CUDA SDK. The main reasons for continuing developing custom hardware are twofold: performance and power consumption. By developing a custom architecture that is fully adapted to a certain range of tasks (as is shown in this chapter), the product of power consumption by performance can be improved by a factor of 100.

3.2 Learning Internal Representations

One of the key questions of Vision Science (natural and artificial) is how to produce good internal representations of the visual world. What sort of internal representation would allow an artificial vision system to detect and classify objects into categories, independently of pose, scale, illumination, conformation, and clutter? More interestingly, how could an artificial vision system *learn* appropriate internal representations automatically, the way animals and humans seem to learn by simply looking at the world? In the time-honored approach to computer vision (and to pattern recognition in general), the question is avoided: internal representations are produced by a hand-crafted feature extractor, whose output is fed to a trainable classifier. While the issue of learning features has been a topic of interest for many years, considerable progress has been achieved in the last few years with the development of so-called *deep learning* methods.

Good internal representations are hierarchical. In vision, pixels are assembled into edglets, edglets into motifs, motifs into parts, parts into objects, and objects into scenes. This suggests that recognition architectures for vision (and for other modalities such as audio and natural language) should have multiple trainable stages stacked on top of each other, one for each level in the feature hierarchy. This raises two new questions: what to put in each stage? and how to train such *deep, multi-stage architectures*? Convolutional Networks (ConvNets) are an answer to the first question. Until recently, the answer to the second question was to use gradient-based supervised learning, but recent research in *deep learning* has produced a number of unsupervised methods which greatly reduce the need for labeled samples.

3.2.1 Convolutional Networks

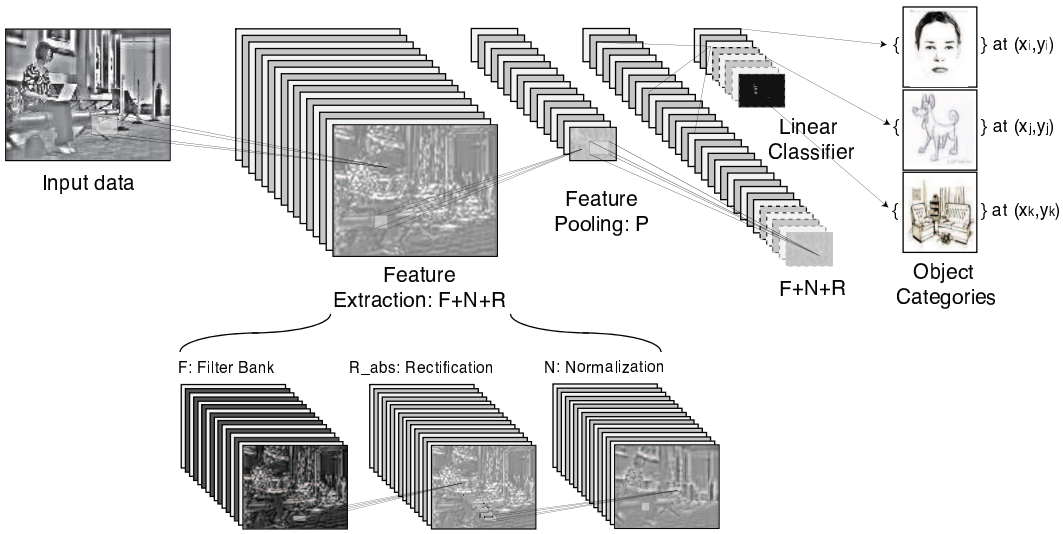

Figure 3.1: Architecture of a typical convolutional network for object recognition. This implements a convolutional feature extractor and a linear classifier for generic N-class object recognition. Once trained, the network can be computed on arbitrary large input images, producing a classification map as output.

Convolutional Networks (64, 65) are trainable architectures composed of multiple stages. The input and output of each stage are sets of arrays called *feature maps*. For example, if the input is a color image, each feature map would be a 2D array containing a color channel of the input image (for an audio input each feature map would be a 1D

array, and for a video or volumetric image, it would be a 3D array). At the output, each feature map represents a particular feature extracted at all locations on the input. Each stage is composed of three layers: a *filter bank layer*, a *non-linearity layer*, and a *feature pooling layer*. A typical ConvNet is composed of one, two or three such 3-layer stages, followed by a classification module.

Each layer type is now described for the case of image recognition. We introduce the following convention: banks of images will be seen as three dimensional arrays in which the first dimension is the number of independent maps/images, the second is the height of the maps and the third is the width. The input bank of a module is denoted x, the output bank y, an image in the input bank x_i, a pixel in the input bank x_{ijk}.

- **Filter Bank Layer -** F: the input is a 3D array with n_1 2D *feature maps* of size $n_2 \times n_3$. Each component is denoted x_{ijk}, and each feature map is denoted x_i. The output is also a 3D array, y composed of m_1 feature maps of size $m_2 \times m_3$. A trainable filter (kernel) k_{ij} in the filter bank has size $l_1 \times l_2$ and connects input feature map x_i to output feature map y_j. The module computes

$$y_j = b_j + \sum_i k_{ij} * x_i \tag{3.1}$$

  where b_j is a trainable bias parameter, and $*$ is the 2D discrete convolution operator:

$$(k_{ij} * x_i)_{pq} = \sum_{m=-l_1/2}^{l_1/2-1} \sum_{n=-l_2/2}^{l_2/2-1} k_{ij,m,n} x_{i,p+m,q+n}. \tag{3.2}$$

  Each filter detects a particular feature at every location on the input. Hence spatially translating the input of a feature detection layer will translate the output but leave it otherwise unchanged.

- **Non-Linearity Layer -** R, N: In traditional ConvNets this simply consists in a pointwise tanh function applied to each site (ijk). However, recent implementations have used more sophisticated non-linearities. A useful one for natural image recognition is the rectified tanh: $R_{abs}(x) = \text{abs}(g_i.tanh(x))$ where g_i is a trainable gain parameter per each input feature map i. The rectified tanh is sometimes

followed by a subtractive and divisive local normalization N, which enforces local competition between adjacent features in a feature map, and between features at the closeby spatial locations. Local competition usually results in features that are decorrelated, thereby maximizing their individual role. The subtractive normalization operation for a given site x_{ijk} computes:

$$v_{ijk} = x_{ijk} - \sum_{ipq} w_{pq}.x_{i,j+p,k+q}, \tag{3.3}$$

where w_{pq} is a normalized truncated Gaussian weighting window (typically of size 9×9). The divisive normalization computes

$$y_{ijk} = \frac{v_{ijk}}{\max(mean(\sigma_{jk}), \sigma_{jk})}, \tag{3.4}$$

where $\sigma_{jk} = (\sum_{ipq} w_{pq}.v^2_{i,j+p,k+q})^{1/2}$. The local contrast normalization layer is inspired by visual neuroscience models (74, 87).

- **Feature Pooling Layer -** P: This layer treats each feature map separately. In its simplest instance, called P_A, it computes the average values over a neighborhood in each feature map. The neighborhoods are stepped by a stride larger than 1 (but smaller than or equal the pooling neighborhood). This results in a reduced-resolution output feature map which is robust to small variations in the location of features in the previous layer. The average operation is sometimes replaced by a max operation, P_M. Traditional ConvNets use a pointwise $tanh()$ after the pooling layer, but more recent models do not. Some ConvNets dispense with the separate pooling layer entirely, but use strides larger than one in the filter bank layer to reduce the resolution (63, 96). In some recent versions of ConvNets, the pooling also pools similar features at the same location, in addition to the same feature at nearby locations (54).

Supervised training is performed using on-line stochastic gradient descent to minimize the discrepancy between the desired output and the actual output of the network. All the coefficients in all the layers are updated simultaneously by the learning procedure for each sample. The gradients are computed with the back-propagation method.

Details of the procedure are given in (65), and methods for efficient training are detailed in (66).

3.2.2 Unsupervised Learning of ConvNets

Training deep, multi-stage architectures using supervised gradient back propagation requires many labeled samples. However in many problems labeled data is scarce whereas unlabeled data is abundant. Recent research in deep learning (7, 48, 88) has shown that *unsupervised learning* can be used to train each stage one after the other using only unlabeled data, reducing the requirement for labeled samples significantly. In (52), using abs and normalization non-linearities, unsupervised pre-training, and supervised global refinement has been shown to yield excellent performance on the Caltech-101 dataset with only 30 training samples per category (more on this below). In (69), good accuracy was obtained on the same set using a very different unsupervised method based on sparse Restricted Boltzmann Machines. Several works at NEC have also shown that using *auxiliary tasks* (4, 103) helps regularizing the system and produces excellent performance.

3.2.2.1 Unsupervised Training with Predictive Sparse Decomposition

The unsupervised method we propose, to learn the filter coefficients in the filter bank layers, is called Predictive Sparse Decomposition (PSD) (53). Similar to the well-known sparse coding algorithms (83), inputs are approximated as a sparse linear combination of dictionary elements.

$$Z^* = \min_Z \|X - WZ\|_2^2 + \lambda |Z|_1 \tag{3.5}$$

In conventional sparse coding (3.5), for any given input X, an expensive optimization algorithm is run to find the optimal sparse representation Z^* (the "basis pursuit" problem). PSD trains a non-linear feed-forward regressor (or *encoder*) $C(X, K) = g.(\tanh(X * k + b))$ to approximate the sparse solution Z^*. During training, the feature vector Z^* is obtained by minimizing the following compound energy:

$$E(Z, W, K) = \|X - WZ\|_2^2 + \lambda \|Z\|_1 + \|Z - C(X, K)\|_2^2 \tag{3.6}$$

where W is the matrix whose columns are the dictionary elements and $K = k, g, b$ are the encoder filter, bias and gain parameters. For each training sample X, one

Table 3.1: Average recognition rates on Caltech-101 with 30 training samples per class. Each row contains results for one of the training protocols (U = unsupervised, X = random, + = supervised fine-tuning), and each column for one type of architecture (F = filter bank, P_A = average pooling, P_M = max pooling, R = rectification, N = normalization).

Single Stage [$64.F^{9\times9} - R/N/P^{5\times5}$ – logreg]				
	$F - R_{abs} - N - P_A$	$F - R_{abs} - P_A$	$F - N - P_M$	$F - P_A$
U^+	54.2%	50.0%	44.3%	14.5%
X^+	54.8%	47.0%	38.0%	14.3%
U	52.2%	43.3%	44.0%	13.4%
X	53.3%	31.7%	32.1%	12.1%
Two Stages [$256.F^{9\times9} - R/N/P^{4\times4}$ – logreg]				
	$F - R_{abs} - N - P_A$	$F - R_{abs} - P_A$	$F - N - P_M$	$F - P_A$
U^+	65.5%	60.5%	61.0%	32.0%
X^+	64.7%	59.5%	60.0%	29.7%
U	63.7%	46.7%	56.0%	9.1%
X	62.9%	33.7%	37.6%	8.8%

first finds Z^* that minimizes E, then W and K are adjusted by one step of stochastic gradient descent to lower E. Once training is complete, the feature vector for a given input is simply approximated with $Z^* = C(X, K)$, hence the process is extremely fast (feed-forward).

3.2.2.2 Results on Object Recognition

In this section, various architectures and training procedures are compared to determine which non-linearities are preferable, and which training protocol makes a difference.

Generic Object Recognition using Caltech 101 Dataset. Caltech 101 is a standard dataset of labeled images, containing 101 categories of objects in the wild.

We use a two-stage system where, the first stage is composed of an F layer with 64 filters of size 9×9, followed by different combinations of non-linearities and pooling. The second-stage feature extractor is fed with the output of the first stage and extracts 256 output features maps, each of which combines a random subset of 16 feature maps from the previous stage using 9×9 kernels. Hence the total number of convolution kernels is $256 \times 16 = 4096$.

Table 3.1 summarizes the results for the experiments, where U and X denotes unsupervised pre-training and random initialization respectively, and $^+$ denotes supervised fine-tuning of the whole system.

1. Excellent accuracy of 65.5% is obtained using unsupervised pre-training and supervised refinement with abs and normalization non-linearities. The result is on par with the popular model based on SIFT and pyramid match kernel SVM (60). It is clear that abs and normalization are crucial for achieving good performance. This is an extremely important fact for users of convolutional networks, which traditionally only use tanh().

2. Astonishingly, *random filters without any filter learning whatsoever achieve decent performance* (62.9% for X), as long as abs and normalization are present ($R_{abs}-N-P_A$). A more detailed study on this particular case can be found in (52).

3. Comparing experiments from rows X vs X^+, U vs U^+, we see that supervised fine tuning consistently improves the performance, particularly with weak non-linearities.

4. It seems that unsupervised pre-training (U, U^+) is crucial when newly proposed non-linearities are not in place.

Handwritten Digit Classification using MNIST Dataset. MNIST is a dataset of handwritten digits (62): it contains $60,000$ 28×28 image patches of digits on uniform backgrounds, and a standard testing set of $10,000$ different samples, widely used by the vision community as a benchmark for algorithms. Each patch is labeled with a number ranging from 0 to 9.

Using the evidence gathered in previous experiments, we used a two-stage system with a two-layer fully-connected classifier to learn the mapping between the samples' pixels and the labels. The two convolutional stages were pre-trained unsupervised (without the labels), and refined supervised (with the labels). An error rate of 0.53% was achieved on the test set. To our knowledge, *this is the lowest error rate ever reported on the original MNIST dataset, without distortions or preprocessing.* The best previously reported error rate was 0.60% (88).

3.2.2.3 Connection with Other Approaches in Object Recognition

Many recent successful object recognition systems can also be seen as single or multi-layer feature extraction systems followed by a classifier. Most common feature extrac-

tion systems like SIFT (73), HoG (28) are composed of filter banks (oriented edge detectors at multiple scales) followed by non-linearities (winner take all) and pooling (histogramming). A Pyramid Match Kernel (PMK) SVM (60) classifier can also be seen as another layer of feature extraction since it performs a K-means based feature extraction followed by local histogramming.

3.3 A Dedicated Digital Hardware Architecture

Biologically inspired vision models, and more generally image processing algorithms are usually expressed as sequences of operations or transformations. They can be well described by a modular approach, in which each module processes an input image bank and produces a new bank. Figure 3.1 is a graphical illustration of this approach. Each module requires the previous bank to be fully (or at least partially) available before computing its output. This causality prevents simple parallelism to be implemented across modules. However parallelism can easily be introduced within a module, and at several levels, depending on the kind of underlying operations.

In the following discussion, banks of images will be seen as three dimensional arrays in which the first dimension is the number of independent maps/images, the second is the height of the maps and the third is the width. As in section 3.2.1, the input bank of a module is denoted x, the output bank y, an image in the input bank x_i, a pixel in the input bank x_{ijk}. Input banks' dimensions will be noted $n_1 \times n_2 \times n_3$, output banks $m_1 \times m_2 \times m_3$. Each module implements a type of operation that requires K operations per input pixel x_{ijk}. The starting point of the discussion is a general purpose processor composed of an arithmetic unit, a fast internal cache of size S_{INT}, and an external memory of size $S_{EXT} >> S_{INT}$. The bandwidth between the internal logic and the external memory array will be noted B_{EXT}.

The coarsest level of parallelism can be obtained at the image bank level. A module that applies a unary transformation to produce one output image for each input image ($n_1 = m_1$) can be broken up in n_1 independent threads. This is the most basic form of parallelism, and it finds its limits when $n_2 \times n_3$ becomes larger than a threshold, closely related to S_{INT}. In fact, past a certain size, the number of pixels that can be processed in a given time equals $B_{EXT}/(2 \times K)$ (bandwidth is shared between writes and reads), assuming that no parallelism is performed at the operation level (the K operations per

pixel are applied sequentially). In other terms, the amount of parallelism that can be introduced at this level is limited by B_{EXT}/K.

A finer level of parallelism can be introduced at the operation level. The cost of fetching pixels from the external memory being very high, the most efficient form of parallelism can occur when pixels are reused in multiple operations ($K > 1$). It can be shown that optimal performances are reached if K operations can be produced in parallel in the arithmetic unit. In other terms, the amount of parallelism that can be introduced at this level is limited by B_{EXT}.

If the internal cache size S_{INT} is large enough to hold all the images of the entire set of modules to compute, then the overall performance of the system if defined by B_{INT}, the bandwidth between the arithmetic unit and the internal cache. As the size of internal memory caches grows (following Moore's law), more data can fit internally, which naturally pushes performance of computations from $K \times B_{EXT}$ to $K \times B_{INT}$.

For a given technology though, S_{INT} has an upper bound, and the only part of the system we can act upon is the internal architecture. Based on these observations, our approach is to tackle the problem of producing the K parallel operations by rethinking the architecture of the arithmetic units, while conserving the traditional external memory storage. Our problem can be stated simply:

Problem 1 *K being the number of operations performed per input pixel; B_{EXT} being the bandwidth available between the arithmetic units and the external memory array; we want to establish an architecture that produces K operations in parallel, so that B_{EXT} is fully utilized.*

3.3.1 A Data-Flow Approach

The data-flow hardware architecture was initiated by (2), and quickly became an active field of research (30, 47, 59). (20) presents one of the latest data-flow architectures that has several similarities to the approach presented here.

Figure 3.2 shows a data-flow architecture whose goal is to process homogeneous streams of data in parallel (32). It is defined around several key ideas:

- a 2D grid of N_{PT} Processing Tiles (PTs) that contain:
  - a bank of processing operators. An operator can be anything from a FIFO to an arithmetic operator, or even a combination of arithmetic operators.

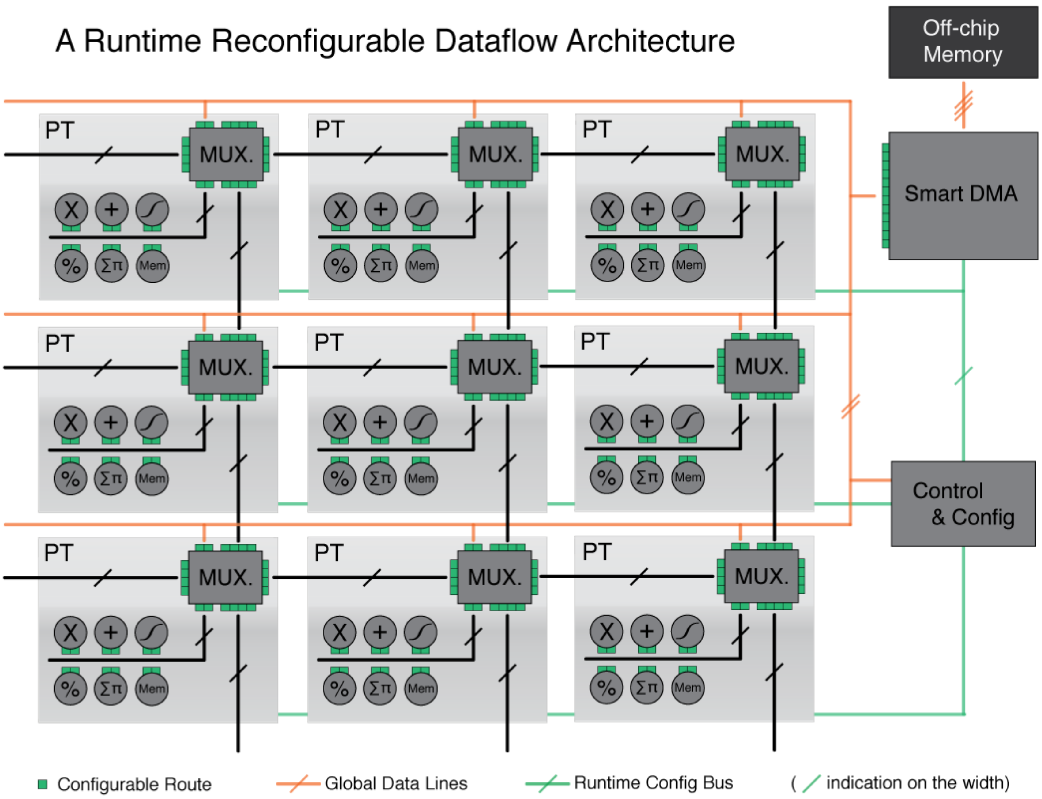

Figure 3.2: A data-flow computer. A set of runtime configurable processing tiles are connected on a 2D grid. They can exchange data with their 4 neighbors and with an off-chip memory via global lines.

The operators are connected to local data lines,

  - a routing multiplexer (MUX). The MUX connects the local data lines to global data lines or to neighboring tiles.

- a Smart Direct Memory Access module (Smart DMA), that interfaces off-chip memory and provides asynchronous data transfers, with priority management,
- a set of N_{global} global data lines used to connect PTs to the Smart DMA, $N_{global} << N_{PT}$,
- a set of local data lines used to connect PTs with their 4 neighbors,
- a Runtime Configuration Bus, used to reconfigure many aspects of the grid at runtime—connections, operators, Smart DMA modes... (the configurable elements are depicted as squares on Fig.3.2),
- a controller that can reconfigure most of the computing grid and the Smart DMA at runtime.

3.3.1.1 On Runtime Reconfiguration

One of the most interesting aspects of this grid is its configuration capabilities. Many systems have been proposed which are based on two-dimensional arrays of processing elements interconnected by a routing fabric that is reconfigurable. Field Programmable Gate Arrays (FPGAs) for instance, offer one of the most versatile grid of processing elements. Each of these processing elements—usually a simple look-up table—can be connected to any of the other elements of the grid, which provides with the most generic routing fabric one can think of. Thanks to the simplicity of the processing elements, the number that can be packed in a single package is in the order of 10^4 to 10^5. The drawback is the reconfiguration time, which takes in the order of milliseconds, and the synthesis time, which takes in the order of minutes to hours depending on the complexity of the circuit.

At the other end of the spectrum, recent multicore processors implement only a few powerful processing elements (in the order of 10s to 100s). For these architectures, no synthesis is involved, instead, extensions to existing programming languages are used to explicitly describe parallelism. The advantage of these architectures is the relative

simplicity of use: the implementation of an algorithm rarely takes more than a few days, whereas months are required for a typical circuit synthesis for FPGAs.

The architecture presented here is at the middle of this spectrum. Building a fully generic data-flow computer is a tedious task. Reducing the spectrum of applications to the image processing problem—as stated in Problem 1—allows us to define the following constraints:

- high throughput is a top priority, low latency is not. Indeed, most of the operations performed on images are replicated over both dimensions of these images, usually bringing the amount of similar computations to a number that is much larger than the typical latencies of a pipelined processing unit,

- therefore each operator has to provide with a maximum throughput (e.g. one operation per clock cycle) to the detriment of any initial latency, and has to be stallable (e.g. must handle discontinuities in data streams).

- configuration time has to be low, or more precisely in the order of the system's latency. This constraint simply states that the system should be able to reconfigure itself between two kinds of operations in a time that is negligible compared to the image sizes. That is a crucial point to allow runtime reconfiguration,

- the processing elements in the grid should be as coarse grained as permitted, to maximize the ratio between *computing logic* and *routing logic*. Creating a grid for a particular application (e.g. ConvNets) allows the use of very coarse operators. On the other hand, a general purpose grid has to cover the space of standard numeric operators,

- the processing elements, although they might be complex, should not have any internal state, but should just passively process any incoming data. The task of sequencing operations is done by a global control unit that simply configures the entire grid for a given operation, lets the data flow in, and prepares the following operation.

The first two points of this list are crucial to create a flexible data-flow system. Several types of grids have been proposed in the past (30, 47, 58), often trying to solve

the dual latency/throughput problem, and often providing a computing fabric that is too rigid.

The grid proposed here provides a flexible processing framework, due to the stallable nature of the operators. Indeed, any paths can be configured on the grid, even paths that require more bandwidth than is actually feasible. Instead of breaking, each operator will stall its pipeline when required. This is achieved by the use of FIFOs at the input and output of each operators, that compensate for bubbles in the data streams, and force the operators to stall when they are full. Any sequence of operators can then be easily created, without concern for bandwidth issues.

The third point is achieved by the use of a runtime configuration bus, common to all units. Each module in the design has a set of configurable parameters, routes or settings (depicted as squares on Figure 3.2), and possesses a unique address on the network. Groups of similar modules also share a broadcast address, which dramatically speeds up reconfiguration of elements that need to perform similar tasks.

The last point depicts the data-flow idea of having (at least theoretically) no state, or instruction pointer. In the case of the system presented here, the grid has no state, but a state does exist in a centralized control unit. For each configuration of the grid, no state is used, and the presence of data drives the computations. Although this leads to an optimal throughput, the system presented here strives to be as general as possible, and having the possibility of configuring the grid quickly to perform a new type of operation is crucial to run algorithms that require different types of computations.

A typical execution of an operation on this system is the following: (1) the control unit configures each tile to be used for the computation and each connection between the tiles and their neighbors and/or the global lines, by sending a configuration command to each of them, (2) it configures the Smart DMA to prefetch the data to be processed, and to be ready to write results back to off-chip memory, (3) when the DMA is ready, it triggers the streaming out, (4) each tile processes its respective incoming streaming data, and passes the results to another tile, or back to the Smart DMA, (5) the control unit is notified of the end of operations when the Smart DMA has completed.

Example 1 *Such a grid can be used to perform arbitrary computations on streams of data, from plain unary operations to complex nested operations. As stated above, operators can be easily cascaded and connected across tiles, independently managing their flow by the use of input/output FIFOs.*

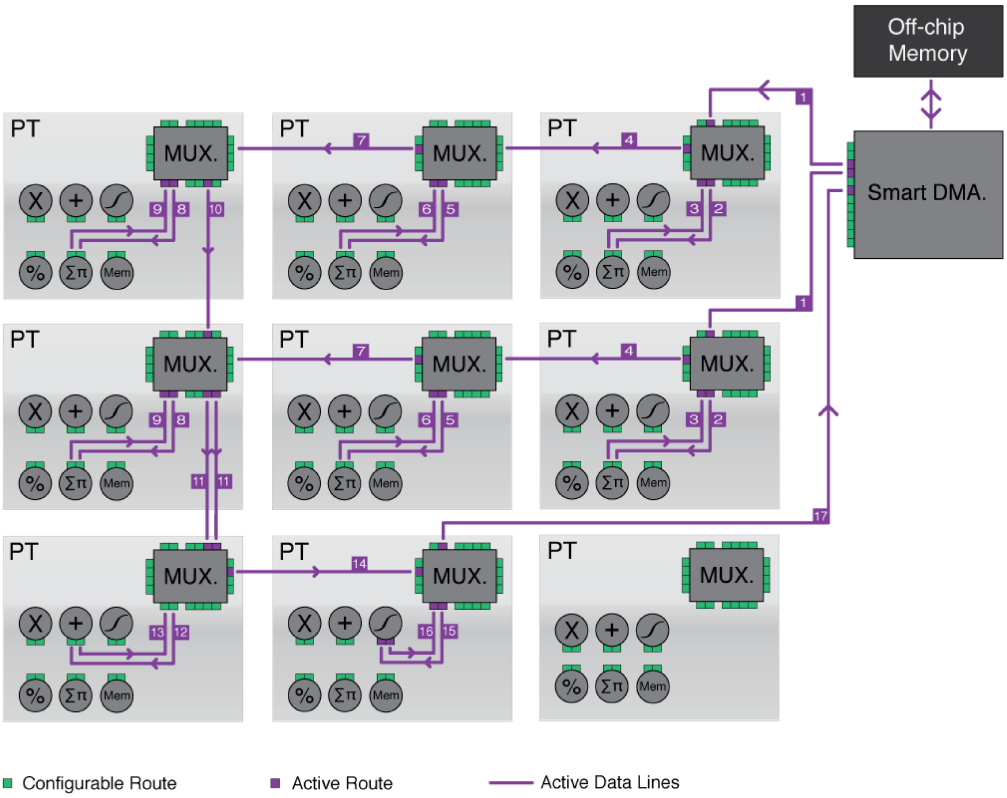

Figure 3.3: The grid is configured for a complex computation that involves several tiles: the 3 top tiles perform a 3×3 convolution, the 3 intermediate tiles another 3×3 convolution, the bottom left tile sums these two convolutions, and the bottom centre tile applies a function to the result.

Figure 3.3 shows an example of configuration, where the grid is configured to compute a sum of two convolutions followed by a non-linear activation function

$$y_{1,i,j} = Tanh(\sum_{m=0}^{K-1}\sum_{n=0}^{K-1} x_{1,i+m,j+n} w_{1,m,n} + \sum_{m=0}^{K-1}\sum_{n=0}^{K-1} x_{2,i+m,j+n} w_{2,m,n}). \tag{3.7}$$

The operator $\sum\prod$ *performs a sum of products, or a dot-product between an incoming stream and a local set of weights (preloaded as a stream too). Therefore each tile performs a 1D convolution, and 3 tiles are used to compute a 2D convolution with a* 3×3 *kernel. All the paths are simplified of course, and in some cases one line represents multiple parallel streams.*

It can be noted that this last example provides a nice solution to Problem 1. Indeed, the input data being 2 images x_1 *and* x_2*, and the output data one image* y_1*, the* K *operations are performed in parallel, and the entire operation is achieved at a bandwidth of* $B_{EXT}/3$*.*

3.3.2 An FPGA-Based ConvNet Processor

Recent DSP-oriented FPGAs include a large number of hard-wired MAC units and several thousands of programmable cells (lookup tables), which allows fast prototyping and real-time simulation of circuits, but also actual implementations to be used in final products.

In this section we present a concrete implementation of the ideas presented in section 3.3.1, specially tailored for ConvNets. We will refer to this implementation as the *Convnet Processor*. The architecture presented here has been fully coded in hardware description languages (HDL) that target both ASIC synthesis and programmable hardware like FPGAs.

A schematic summary of the *ConvNet Processor* system is presented in Figure 3.4. The main components of our system are: (1) a *Control Unit* (implemented on a general purpose CPU), (2) a grid of *Processing Tiles (PTs)*, and (3) a *Smart DMA* interfacing external memory via a standard controller.

In this implementation, the Control Unit is implemented by a *general purpose CPU*. This is more convenient than a custom state machine as it allows the use of standard C compilers. Moreover, the CPU has full access to the external memory (via global data lines), and it can use this large storage to store its program instructions.

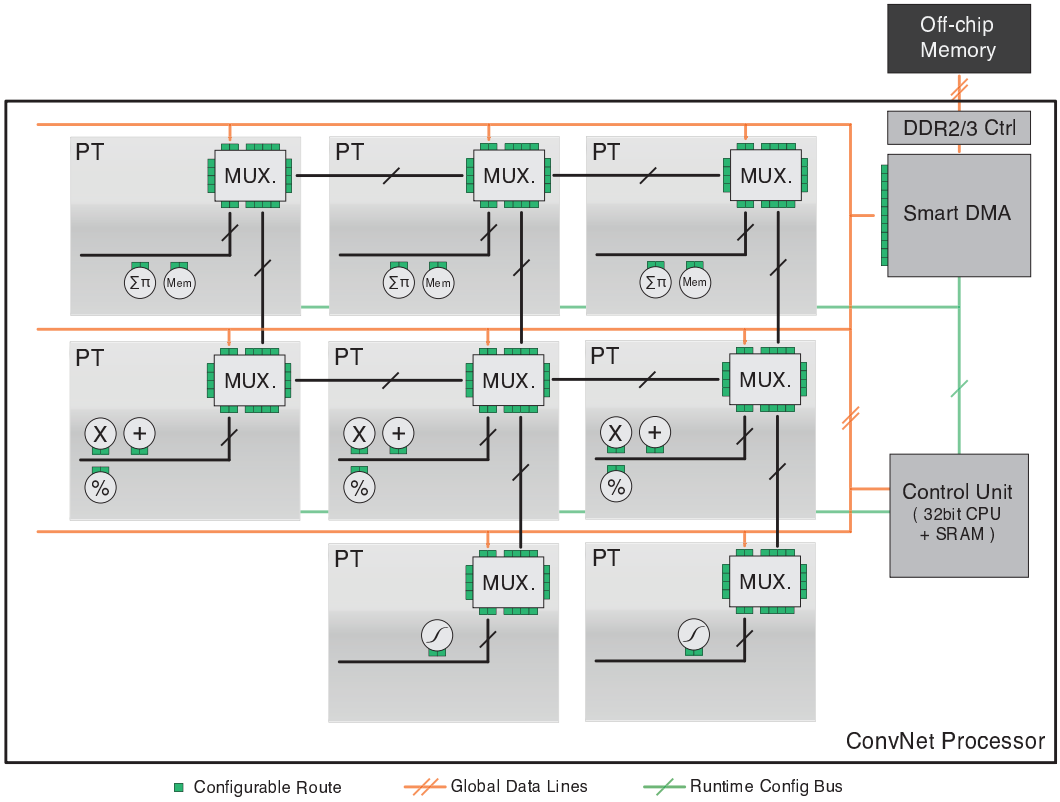

Figure 3.4: Overview of the ConvNet Processor system. A grid of multiple full-custom Processing Tiles tailored to ConvNet operations, and a fast streaming memory interface (Smart DMA).

3.3.2.1 Specialized Processing Tiles

The *PTs* are independent processing tiles laid out on a two-dimensional grid. As presented in section 3.3.1, they contain a routing multiplexer (MUX) and local operators. Compared to the general purpose architecture proposed above, this implementation is specialized for ConvNets and other applications that rely heavily on two-dimensional convolutions (from 80% to 90% of computations for ConvNets).

Figure 3.4 shows this specialization:

- the top row PTs only implement Multiply and Accumulate (MAC) arrays ($\sum\prod$ operators), which can be used as 2D convolvers (implemented in the FPGA by dedicated hardwired MACs). It can also perform on-the-fly subsampling (spatial pooling), and simple dot-products (linear classifiers) (31),

- the middle row PTs contain general purpose operators (squaring and dividing are necessary for divisive normalization),

- the bottom row PTs implement non-linear mapping engines, used to compute all sorts of functions from $Tanh()$ to $Sqrt()$ or $Abs()$. Those can be used at all stages of the ConvNets, from normalization to non-linear activation units.

The operators in the PTs are fully pipelined to produce one result per clock cycle. Image pixels are stored in off-chip memory as Q8.8 (16bit, fixed-point), transported on global lines as Q8.8 but scaled to 32bit integers within operators, to keep full precision between successive operations. The numeric precision, and hence the size of a pixel, will be noted P_{bits}.

The 2D convolver can be viewed as a data-flow grid itself, with the only difference that the connections between the operators (the MACs) are fixed. The reason for having a full-blown 2D convolver within a tile (instead of a 1D convolver per tile, or even simply one MAC per tile) is that it maximizes the ratio between actual computing logic and routing logic, as stated previously. Of course it is not as flexible, and the choice of the array size is a hardwired parameter, but it is a reasonable choice for an FPGA implementation, and for image processing in general. For an ASIC implementation, having a 1D dot-product operator per tile is probably the best compromise.

The pipelined implementation of this 2D convolver (as described in (31)), computes Equation 3.8 at every clock cycle.

$$y_{1,i,j} = x_{2,i,j} + \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} x_{1,i+m,j+n} w_{1,m,n} \tag{3.8}$$

In equation 3.8 $x_{1,i,j}$ is a value in the input plane, $w_{1,m,n}$ is a value in a $K \times K$ convolution kernel, $x_{2,i,j}$ is a value in a plane to be combined with the result, and y_1 is the output plane.

Both the kernel and the image are streams loaded from the memory, and the filter kernels can be pre-loaded in local caches concurrently to another operation: each new pixel thus triggers $K \times K$ parallel operations.

All the non-linearities in neural networks can be computed with the use of look-up tables or piece-wise linear decompositions.

A loop-up table associates one output value for each input value, and therefore requires as much memory as the range of possible inputs. It is the fastest method to compute a non-linear mapping, but the time required to reload a new table is prohibitive if different mappings are to be computed with the same hardware (and the memory required can be prohibitive as well).

A piece-wise linear decomposition is not as accurate (f is approximated by g, as in Eq. 3.9), but only requires a couple of coefficients a_i to represent a simple mapping such as a hyperbolic tangent, or a square root (for a limited memory budget, it is therefore more accurate than a look-up table). It can be reprogrammed very quickly at runtime, allowing multiple mappings to reuse the same hardware. Moreover, if the coefficients a_i follow the constraint given by Eq. 3.10, the hardware can be reduced to shifters and adders only (divisions by a powers of 2).

$$g(x) = a_i x + b_i \quad for \quad x \in [l_i, l_{i+1}] \tag{3.9}$$

$$a_i = \frac{1}{2^m} + \frac{1}{2^n} \quad m, n \in [0, 5]. \tag{3.10}$$

3.3.2.2 Smart DMA Implementation

A critical part of this architecture is the Direct Memory Access (DMA) module. Our *Smart DMA* module is a full custom engine that has been designed to allow N_{DMA} ports to access the external memory totally asynchronously.

A dedicated arbiter is used as hardware *Memory Interface* to multiplex and demultiplex access to the external memory with high bandwidth. Subsequent buffers on each port insure continuity of service on a port while the others are utilized.

The DMA is *smart*, because it complements the Control Unit. Each port of the DMA can be configured to read or write a particular chunk of data, with an optional stride (for 2D streams), and communicate its status to the Control Unit. Although this might seem trivial, it respects one the foundations of data-flow computing: while the Control Unit configures the grid and the DMA ports for each operation, an operation is driven exclusively by the data, from its fetching, to its writing back to off-chip memory.

If the PTs are synchronous to the memory bus clock, the following relationship can be established between the memory bandwidth B_{EXT}, the number of possible parallel data transfers $MAX(N_{DMA})$ and the bits per pixel P_{bits}:

$$MAX(N_{DMA}) = \frac{B_{EXT}}{P_{bits}}. \tag{3.11}$$

For example $P_{bits} = 16$ and $B_{EXT} = 128bit/cyc$ allows $MAX(N_{DMA}) = 7$ simultaneous transfers.

3.3.3 Compiling ConvNets for the ConvNet Processor

Prior to being run on the ConvNet Processor, a ConvNet has to be trained offline, on a regular computer, and then converted to a compact representation that can be interpreted by the Control Unit to generate controls/configurations for the system.

Offline, the training is performed with existing software such as Lush (61) or Torch-5 (23). Both libraries use the modular approach described in the introduction of section 3.3.

On board, the Control Unit of the ConvNet Processor decodes the representation, which results in several grid reconfigurations, interspersed with data streams. This representation will be denoted as *bytecode* from now on. Compiling a ConvNet for the ConvNet Processor can be summarized as the task of mapping the offline training results to this bytecode.

Extensive research has been done on the question of how to schedule data-flow computations (68), and how to represent streams and computations on streams (59).

In this section, we only care about how to schedule computations for a ConvNet (and similar architectures) on our ConvNet Processor engine.

It is a more restricted problem, and can be stated simply:

Problem 2 *Given a particular ConvNet architecture, and trained parameters, and given a particular implementation of the data-flow grid, what is the sequence of grid configurations that yield the shortest computation time? Or in other terms, for a given ConvNet architecture, and a given data-flow architecture, how to produce the bytecode that yields the shortest computing time?*

As described in the introduction of section 3.3, there are three levels at which computations can be parallelized:

- across modules: operators can be cascaded, and multiple modules can be computed on the fly (average speedup),
- across images, within a module: can be done if multiple instances of the required operator exist (poor speedup, as each independent operation requires its own input/output streams, which are limited by B_{EXT}),
- within an image: some operators naturally implement that (the 2D convolver, which performs all the MACs in parallel), in some cases, multiple tiles can be used to parallelize computations.

Parallelizing computations across modules can be done in special cases. Example 1 illustrates this case: two operators (each belonging to a separate module) are cascaded, which speeds up this computation by a factor of 2.

Parallelizing computations across images is straightforward but very limited. Here is an example that illustrates that point:

Example 2 *The data-flow system built has 3 PTs with 2D convolvers, 3 PTs with standard operators, and 2 PTs with non-linear mappers (as depicted in Figure 3.4, and the exercise is to map a fully-connected filter-bank with 3 inputs and 8 outputs, e.g. a filer bank where each of the 8 outputs is a sum of 3 inputs convolved with a different kernel:*

$$y_j = \sum_{i=0}^{2} k_{ij} * x_i \quad for \quad j \in [0, 7]. \tag{3.12}$$

For the given hardware, the optimal mapping is: each of the three 2D convolvers is configured to convolve one of the three inputs x_i with a kernel k_{ij}, and a standard PT is configured to accumulate those 3 streams in one and produce y_j.

Although optimal (3 images are processed in parallel), 4 simultaneous streams are created at the Smart DMA level, which imposes a maximum bandwidth of $B_{EXT}/4$ per stream.

Parallelizing computations within images is what this grid is best at. Example 1 is a perfect example of how an operation (in that case a sequence of operations) can be done in a single pass on the grid.

3.3.4 Application to Scene Understanding

Several applications were implemented on neuFlow: from a simple face detector to a pixel-wise obstacle classifier (25) and a complete street scene parser, as shown on Figure 3.5. Other example applications can be found at `www.neuflow.org`.

In this section we focus on the elaboration, training and implementation of a complete street-scene parser. This work extends and is strongly inspired by previous work from Grangier *et al.* (43). Scene parsing, as seen in Chapter 2, aims at segmenting and recognizing the content of a scene: from objects to large structures—roads, sky, buildings, cars, etc. In other words, the goal is to map each pixel of a given input image to a unique label.

Grangier *et al.* (43) showed that using a deep convolutional network with a greedy layer-wise learning (up to 6 convolutional layers) could yield significantly better results than simpler 2 or 3-layer systems. In Chapter 2, we showed how a multiscale network could efficiently encode and describe image patches for this type of task. We followed a slightly different method here, favoring larger kernels over deeper networks, as these are easily accelerated with our hardware, but kept the idea of incrementally growing the network's capacity.

A subset of the LabelMe dataset (91), containing about 3000 images of spanish cities [1], was used to train this convolutional network. We removed 10% of the set to be used for validation (testing). The twenty most occurring classes were extracted, and the goal was set to minimize the pixel classification error on those classes.

[1] `http://people.csail.mit.edu/torralba/benchmarks/`

Figure 3.5: Scene Parsing on FPGAs - Street scene parsing: a convolutional network was trained on the LabelMe spanish dataset (91) with a method similar to (43). The training set only contains photos from spanish cities; the image above is a picture taken in Edinburgh. The convolutional network is fully computed on neuFlow, achieving a speedup of about 100x (500x375 images are processed in 83ms, as opposed to 8s on a laptop).

All the images were first resized to 500×375, then 400 million patches were randomly sampled to produce a $20 \times 1e8 \times N \times N$ tensor where the first dimension indexes the classes, the second indexes patches of which the center pixel belongs to the corresponding class, and the last two dimensions are the height and width of the patch.

The training was done in 3 phases. First: we started with a simple model, CN_1 (table 3.2), similar to the one originally proposed in (65). The model has small kernels (5×5) and 3 convolutional layers only. This first model was trained to optimize the pixel-wise cross entropy (negative log-likelihood) through stochastic gradient descent over the training set. Minimizing the cross entropy (rather than the mean-square error) helps promote the categories of rare appearance. Small kernels, and a few layers allowed the system to see 10 million training patches in a couple of hours, and converge to a reasonable error fairly quickly. With these parameters, the receptive field of the network is 32×32, which only represents 0.55% of the complete field of view;

Second: all the convolutional kernels were then increased to 9×9, by padding the extra weights with zeros: CN_2 (table 3.3). This increased the receptive field to 60×60 (about 2% of the image), with the interesting property that at time 0 of this second training phase, the network was producing the same predictions than with the smaller kernels;

Third: a fourth layer was added—a.k.a. greedy layer-wise learning—which increased the receptive field to 92×92 (5% of the image). This required dropping the previous linear classifier, and replace it with a new—randomly initialized—larger classifier.

Performances were evaluated on a separate test set, which was created using a subset (10%) of the original dataset. Results are shown on Table 3.5.

Once trained, the network was passed over to luaFlow, and transparently mapped to neuFlow. A key advantage of convolutional networks is that they can be applied to sliding windows on a large image at very low cost by simply computing convolutions at each layer over the entire image. The output layer is replicated accordingly, producing one detection score for every 92×92 window on the input, spaced every 4 pixels. Producing the prediction on one image of that size takes about 8 seconds on a laptop-class Intel DuoCore 2.66GHz processor; the same prediction is produced in $83ms$ on neuFlow, with an average error of 10^{-2} (quantization noise).

Layer	Kernels: dims [nb]	Maps: dims [nb]
Input image		32×32 [3]
N0 (Norm)		32×32 [3]
C1 (Conv)	5×5 [48]	28×28 [12]
P2 (Pool)	2×2 [1]	14×14 [12]
C3 (Conv)	5×5 [384]	10×10 [32]
P4 (Pool)	2×2 [1]	5×5 [32]
C5 (Conv)	5×5 [1536]	1×1 [48]
L (Linear)	1×1 [960]	1×1 [20]

Table 3.2: CN_1: base model. N: Local Normalization layer (note: only the Y channel is normalized, U and V are untouched); C: convolutional layer; P: pooling (max) layer; L: linear classifier.

3.3.5 Performance

Figure 3.6 reports a performance comparison for the computation of a typical ConvNet on multiple platforms:

- the CPU data was measured from compiled C code (GNU C compiler and Blas libraries) on a Core 2 Duo 2.66GHz Apple Macbook PRO laptop operating at 90W (30 to 40W for the CPU);
- the FPGA data was measured on both a Xilinx Virtex-4 SX35 operating at 200MHz and 7W and a Xilinx Virtex-6 VLX240T operating at 200MHz and 10W;
- the GPU data was obtained from a CUDA-based implementation running on a laptop-range nVidia GT335m operating at 1GHz and 40W;
- the ASIC data is simulation data gathered from an IBM 65nm CMOS process.For an ASIC-based design with a speed of 400MHz (speeds of up to > 1 GHz are possible), the projected power consumption is simulated at 3W.

The test ConvNet is composed of a non-linear normalization layer, 3 convolutional layers, 2 pooling layers, and a linear classifier. The convolutional layers and pooling layers are followed by non-linear activation units (hyperbolic tangent). Overall, it

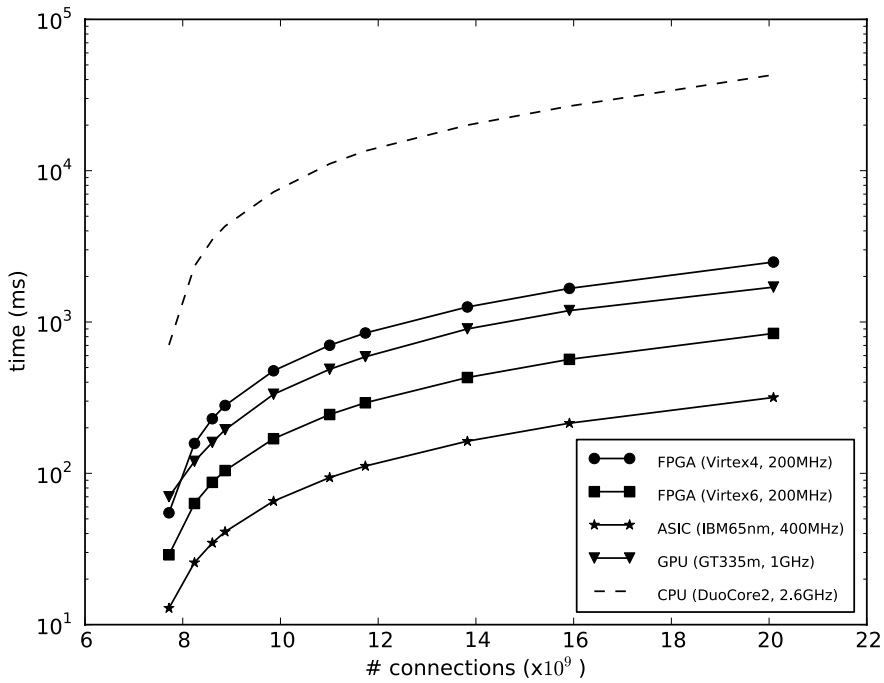

Figure 3.6: Compute time for a typical ConvNet (as seen in Figure 3.1).

Layer	Kernels: dims [nb]	Maps: dims [nb]
Input image		60×60 [3]
N0 (Norm)		60×60 [3]
C1 (Conv)	9×9 [48]	52×52 [12]
P2 (Pool)	2×2 [1]	26×26 [12]
C3 (Conv)	9×9 [384]	18×18 [32]
P4 (Pool)	2×2 [1]	9×9 [32]
C5 (Conv)	9×9 [1536]	1×1 [48]
L (Linear)	1×1 [960]	1×1 [20]

Table 3.3: CN_2: second model. Filters are increased, which doubles the receptive field

possesses N_{KER} $K \times K$ learned kernels, N_{POOL} $P \times P$ learned pooling kernels, and N 200 dimension classification vectors.

Figure 3.6 was produced by increasing the parameters N_{KER}, N_{POOL}, K and P simultaneously, and estimating the time to compute the ConvNet for each set of parameters. The x-axis reports the overall number of linear connections in the ConvNet (e.g. the number of multiply and accumulate operations to perform).

Note: on the spectrum of parallel computers described in Section 3.3.1.1, GPUs belong to the small grids (100s of elements) of large and complex processing units (full-blown streaming processors). Although they offer one of the most interesting ratio of computing power over price, their drawback is their high power consumption (from 40W to 200W per unit).

Table 3.6 reports a performance comparison for the computation of a typical filter bank operation on multiple platforms: 1- the CPU data was measured from compiled C code (GNU C compiler and Blas libraries) on a Core 2 Duo 2.66GHz Apple Macbook PRO laptop operating at 90W (30W for the CPU); 2- the FPGA data was measured on a Xilinx Virtex-6 VLX240T operating at 200MHz and 10W (power consumption was measured on the board) ; 3- the GPU data was obtained from a CUDA-based implementation running on a laptop-range nVidia GT335m operating at 1GHz and 30W and on a nVidia GTX480 operating at 1GHz and 220W; 4- the ASIC data is simulation data gathered from an IBM 45nm CMOS process ($5 \times 5mm$). For an ASIC-based design with a speed of 400MHz, the projected power consumption, using post-synthesis data

Layer	**Kernels: dims [nb]**	**Maps: dims [nb]**
Input image		92×92 [3]
N0 (Norm)		92×92 [3]
C1 (Conv)	9×9 [48]	84×84 [12]
P2 (Pool)	2×2 [1]	42×42 [12]
C3 (Conv)	9×9 [384]	34×34 [32]
P4 (Pool)	2×2 [1]	17×17 [32]
C5 (Conv)	9×9 [1536]	9×9 [48]
C6 (Conv)	9×9 [1024]	1×1 [128]
L (Linear)	1×1 [960]	1×1 [20]

Table 3.4: CN_3: a fourth convolutional layer C6 is added, which, again, increases the receptive field. Note: C6 has sparse connectivity (*e.g.*each of its 128 outputs is connected to 8 inputs only, yielding 1024 kernels instead of 6144).

Model	CN_1	CN_2	CN_3
CN Error (%)	29.75	26.13	24.26
CN+MST Error (%)	27.17	24.40	23.39

Table 3.5: Percentage of mislabeled pixels on validation set. CN Error is the pixelwise error obtained when using the simplest pixelwise winner, predicted by the ConvNet. CN+MST Error is the pixelwise error obtained by histogramming the ConvNet's prediction into connected components (the components are obtained by computing the minimum spanning tree of an edge-weighted graph built on the raw RGB image, and merging its nodes using a surface criterion, in the spirit of (35)).

and standard analysis tools is estimated at 5W.

The current design was proven at 200MHz on a Xilinx Virtex 6 ML605 platform, using four 10×10 convolver grids. At this frequency, the peak performance is 80 billion connections per second, or 160 GOPs. Sustained performances for typical applications (such as the street scene parser) range from 60 to 120 GOPs, sustained.

3.3.6 Precision

Recognition rates for standard datasets were obtained to benchmark the precision loss induced by the fixed-point coding. Using floating-point representation for training

	CPU	V6	mGPU	IBM	GPU
Peak GOPs	10	160	182	1280	1350
Real GOPs	1.1	147	54	1164	294
Power W	30	10	30	5	220
GOPs/W	0.04	14.7	1.8	230	1.34

Table 3.6: Performance comparison. 1- CPU: Intel DuoCore, 2.7GHz, optimized C code, 2- V6: neuFlow on Xilinx Virtex 6 FPGA—on board power and GOPs measurements; 3- IBM: neuFlow on IBM 45nm process: simulated results, the design was fully placed and routed; 4- mGPU/GPU: two GPU implementations, a low power GT335m and a high-end GTX480.

and testing, the following results were obtained: for *NORB*, 85% recognition rate was achieved on the test dataset, for *MNIST*, 95% and for *UMASS* (faces dataset), 98%. The same tests were conducted on the ConvNet Processor with fixed-point representation (Q8.8), and the results were, respectively: 85%, 95% and 98%, which confirms the assumptions made a priori on the influence of quantization noise.

To provide more insight into the fixed-point conversion, the number of weights being zeroed with quantization was measured, in the case of the NORB object detector. Figure 3.7 shows the results: at 8bits, the quantization impact is already significant (10% of weights become useless), although it has no effect on the detection accuracy.

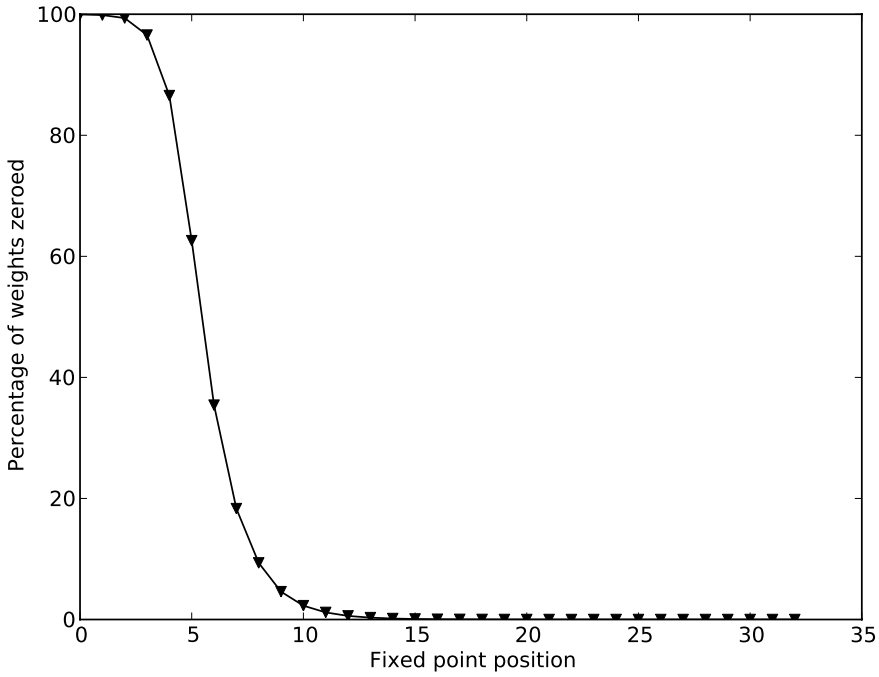

Figure 3.7: Quantization effect on trained networks: the x axis shows the fixed point position, the y axis the percentage of weights being zeroed after quantization.

4

Discussion

In this thesis I presented three contributions: (1) a multiscale deep convolutional network architecture to easily capture long-distance relationships between input variables in image data, (2) a tree-based algorithm to efficiently explore multiple segmentation candidates, to produce maximally confident semantic segmentations of images, (3) a custom dataflow computer architecture optimized for the computation of convolutional networks, and similarly dense image processing models. All three contributions were produced with the common goal of getting us closer to real-time image understanding.

Contribution (1) was deployed in production at a company I co-founded called MadBits [1]. They are now part of a larger framework that relies on deep networks to learn rich representations of images, and enable a wide range of features: classification, text-based search, image-based search (search in model's feature space), online learning (using density estimation in feature space), ...

My co-author Camille Couprie extended this work in (27) by applying it to RGB-D imagery (image+depth data), to solve the problem of indoor semantic segmentation. The overall method produced state-of-the-art results on a standard benchmark.

Contribution (3) was used to power several DARPA and ONR projects, and was successfully integrated into industry-level systems by HRL Laboratories. It was also tested by multiple companies and research laboratories. In particular, it was fully implemented as an ASIC, 45nm, IBM technology (86).

Contribution (3) also served as the technological basis for another company called

[1] `http://www.madbits.com`

TeraDeep [1], which focuses on building high-performance hardware for deep-learning based applications.

There are several extensions and future directions that shouwld follow this thesis.

Contribution (1) can easily be extended to work on all sorts of other modalities: video, speech, music... The idea of a coarse-to-fine, or simply multiscale representation with weight-sharing is fairly generic, and can be seen as a powerful regularizer that exploits the fact that all these signals can be generated at arbitrary scales, such that features benefit from being learned and extracted at multiple scales. Next, a more unsupervised approach to feature learning could potentially help produce more generic features, by exploiting much larger amounts of unlabeled data (obtaining labeled data for semantic segmentation tasks is very expensive). This was largely unexplored during this thesis, mostly because the current benefits of unsupervised pre-training are still negligible, when compared to the gains obtained with a better model architecture and/or increased amounts of labeled data.

Contribution (2) can also naturally be applied to other modalities, and in particular can easily be extended to video data. To do so, the segmentation tree must simply be constructed on volumes of pixels (by constructing a graph over pixels in space and time). Couprie *et al.* (26) proposed a causal graph-based video segmentation which is perfectly fit for this type of task, and has the benefit of being causal (which means it can be used to process real-time video streams).

Contribution (3) was mostly limited by in-chip memory at the time of this thesis. The greatest gains will be achieved by migrating most of the off-chip storage to in-chip, distributed memory, which will enable one to two orders of magnitude improvement in both power consumption and processing speed. In a foreseeable future, generic, programmable, and self-contained circuits will be added to industry-standard processors, especially mobile ones, to enable efficient computations of basic deep network operators. This will enable a wide range of recognition applications to be embedded in cheap mobile devices.

[1] `http://www.teradeep.com`

References

[1] Abdulkader, A. (2006). A two-tier approach for arabic offline handwriting recognition. In *IWFHR'06*. 8

[2] Adams, D.A. (1969). *A computation model with data flow sequencing*. Ph.D. thesis, Stanford, CA, USA. 66

[3] Adé, M., Lauwereins, R. & Peperstraete, J. (1997). Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets. In *Proceedings of the 34th annual Design Automation Conference*, 64–69, ACM. 21

[4] Ahmed, A., Yu, K., Xu, W., Gong, Y. & Xing, E. (2008). Training hierarchical feed-forward visual recognition models using transfer learning from pseudo-tasks. In *ECCV*, Springer-Verlag. 58, 62

[5] Anant Agarwal, J.B.B.E.M.M.C.C.M.C.R.D.W., Liewei Bao (2007). Tile processor: Embedded multicore for networking and multimedia. *Hotchips*. 21

[6] Arbeláez, P., Maire, M., Fowlkes, C. & Malik, J. (2011). Contour Detection and Hierarchical Image Segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, **33**, 898–916. 38, 52

[7] Bengio, Y., Lamblin, P., Popovici, D. & Larochelle, H. (2007). Greedy layer-wise training of deep networks. In *NIPS*. 9, 62

[8] Berg, A.C., Berg, T.L. & Malik, J. (2005). Shape matching and object recognition using low distortion correspondences. In *CVPR*. 57

[9] Bhattacharyya, S.S., Murthy, P.K. & Lee, E.A. (1996). *Software synthesis from dataflow graphs*, vol. 360. Springer. 21

[10] Bhattacharyya, S.S., Murthy, P.K. & Lee, E.A. (1999). Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, **21**, 151–166. 21

[11] Bottou, L. (1998). Online algorithms and stochastic approximations. In D. Saad, ed., *Online Learning and Neural Networks*, Cambridge University Press, Cambridge, UK, revised, oct 2012. 12

[12] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Y. Lechevallier & G. Saporta, eds., *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, 177–187, Springer, Paris, France. 12

[13] Bottou, L. & LeCun, Y. (2004). Large scale online learning. In S. Thrun, L. Saul & B. Schölkopf, eds., *Advances in Neural Information Processing Systems 16*, MIT Press, Cambridge, MA. 12

[14] Boykov, Y. & Jolly, M.P. (2001). Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images. In *Proceedings of International Conference of Computer Vision (ICCV)*, vol. 1, 105–112. 46

[15] Boykov, Y. & Kolmogorov, V. (2004). An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, **26**, 1124–1137. 36

[16] Boykov, Y., Veksler, O. & Zabih, R. (2001). Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, **23**, 1222–1239. 36

[17] Chellapilla, K. & Simard, P. (2006). A new radical based approach to offline handwritten east-asian character recognition. In *IWFHR*. 8

[18] Chellapilla, K., Puri, S. & Simard, P. (2006). High performance convolutional neural networks for document processing. In *IWFHR*. 8

[19] Chellapilla, K., Shilman, M. & Simard, P. (2006). Optimally combining a cascade of classifiers. In *Proc. of Document Recognition and Retrieval 13, Electronic Imaging, 6067*. 8

[20] Cho, M.H., chi Cheng, C., Kinsy, M., Suh, G.E. & Devadas, S. (2008). Diastolic arrays: Throughput-driven reconfigurable computing. 66

[21] Ciresan, D., Meier, U., Masci, J. & Schmidhuber, J. (2011). A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, 1918–1921, IEEE. 45

[22] Coates, A., Baumstarck, P., Le, Q. & Ng, A. (2009). Scalable learning for object detection with gpu hardware. In *Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, 4287–4293, Citeseer. 58

[23] Collobert, R. (2008). Torch. presented at the Workshop on Machine Learning Open Source Software, NIPS. 76

[24] Collobert, R. (2011). Deep learning for efficient discriminative parsing. *VideoLectures.net*, 7:45. 3

[25] Corda, B., Farabet, C., Scoffier, M. & LeCun, Y. (2010). Building heterogeneous platforms for end-to-end online learning based on dataflow computing design. 78

[26] Couprie, C., Farabet, C. & LeCun, Y. (2013). Causal graph-based video segmentation. In *ArXiv*. 88

[27] Couprie, C., Farabet, C., Najman, L. & LeCun, Y. (2013). Indoor semantic segmentation using depth information. In *Proceedings of the International Conference on Learning Representations*. 87

[28] Dalal, N. & Triggs, B. (2005). Histograms of oriented gradients for human detection. In *CVPR*. 5, 57, 65

[29] Delakis, M. & Garcia, C. (2008). Text detection with convolutional neural networks. In *International Conference on Computer Vision Theory and Applications (VISAPP 2008)*. 8

[30] Dennis, J.B. & Misunas, D.P. (1974). A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, **3**, 126–132. 66, 69

[31] Farabet, C., Poulet, C., Han, J.Y. & LeCun, Y. (2009). Cnp: An fpga-based processor for convolutional networks. In *International Conference on Field Programmable Logic and Applications (FPL'09)*, IEEE, Prague. 74, 75

[32] Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y. & Culurciello, E. (2010). Hardware accelerated convolutional neural networks for synthetic vision systems. In *International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, Paris. 48, 49, 66

[33] Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E. & LeCun, Y. (2011). Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the Fifth IEEE Workshop on Embedded Computer Vision*, IEEE. 48, 49

[34] Farabet, C., Couprie, C., Najman, L. & LeCun, Y. (2012). Scene parsing with multiscale feature learning, purity trees, and optimal covers. In *Proceedings of the International Conference on Machine Learning (ICML)*. 29, 38

[35] Felzenszwalb, P. & Huttenlocher, D. (2004). Efficient graph-based image segmentation. *International Journal of Computer Vision*, **59**, 167–181. xiii, 17, 33, 38, 42, 44, 46, 84

[36] Ford, L.R. & Fulkerson, D.R. (1955). A simple algorithm for finding maximal network flows and an application to the hitchcock problem. Tech. rep., RAND Corp., Santa Monica. 46

[37] Frome, A., Cheung, G., Abdulkader, A., Zennaro, M., Wu, B., Bissacco, A., Adam, H., Neven, H. & Vincent, L. (2009). Large-scale privacy protection in street-level imagery. In *ICCV'09*. 8

[38] Fukushima, K. & Miyake, S. (1982). Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, **15**, 455–469. 8

[39] Fulkerson, B., Vedaldi, A. & Soatto, S. (2009). Class segmentation and object localization with superpixel neighborhoods. In *ICCV*, 670–677, IEEE. 33, 35

[40] Garcia, C. & Delakis, M. (2004). Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 8

[41] Gould, S., Rodgers, J., Cohen, D., Elidan, G. & Koller, D. (2008). Multi-class segmentation with relative location prior. *Int. J. Comput. Vision*, **80**, 300–316. 33

[42] Gould, S., Fulton, R. & Koller, D. (2009). Decomposing a scene into geometric and semantically consistent regions. *IEEE International Conference on Computer Vision*, 1–8. 41, 42

[43] Grangier, D., Bottou, L. & Collobert, R. (2009). Deep convolutional networks for scene parsing. ICML 2009 Deep Learning Workshop. 78, 79

[44] Hadsell, R., Chopra, S. & LeCun, Y. (2006). Dimensionality reduction by learning an invariant mapping. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*, IEEE Press. 47

[45] Hadsell, R., Sermanet, P., Scoffier, M., Erkan, A., Kavackuoglu, K., Muller, U. & LeCun, Y. (2009). Learning long-range vision for autonomous off-road driving. *Journal of Field Robotics*, **26**, 120–144. 9

[46] Happold, M. & Ollis, M. (2007). Using learned features from 3d data for robot navigation. 9

[47] Hicks, J., Chiou, D., Ang, B.S. & Arvind (1993). Performance studies of id on the monsoon dataflow system. 66, 69

[48] Hinton, G.E. & Salakhutdinov, R.R. (2006). Reducing the dimensionality of data with neural networks. *Science*. 9, 62

[49] Huang, F.J. & LeCun, Y. (2006). Large-scale learning with svm and convolutional nets for generic object categorization. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*, IEEE Press. 57

[50] Jain, V. & Seung, H.S. (2008). Natural image denoising with convolutional networks. In *Advances in Neural Information Processing Systems 21 (NIPS 2008)*, MIT Press. 9

[51] Jain, V., Murray, J.F., Roth, F., Turaga, S., Zhigulin, V., Briggman, K., Helmstaedter, M., Denk, W. & Seung, S.H. (2007). Supervised learning of image restoration with convolutional networks. In *ICCV*. 9

[52] Jarrett, K., Kavukcuoglu, K., Ranzato, M. & LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, IEEE. 51, 58, 62, 64

[53] Kavukcuoglu, K., Ranzato, M. & LeCun, Y. (2008). Fast inference in sparse coding algorithms with applications to object recognition. Tech. rep., tech Report CBLL-TR-2008-12-01. 10, 51, 62

[54] Kavukcuoglu, K., Ranzato, M., Fergus, R. & LeCun, Y. (2009). Learning invariant features through topographic filter maps. In *Proc. International Conference on Computer Vision and Pattern Recognition*, IEEE. 8, 51, 58, 61

[55] Kavukcuoglu, K., Sermanet, P., Boureau, Y., Gregor, K., Mathieu, M. & LeCun, Y. (2010). Learning convolutional feature hierachies for visual recognition. In *Advances in Neural Information Processing Systems (NIPS 2010)*, vol. 23. 51

[56] Kruskal, J. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the AMS*, 48–50. 15

[57] Kumar, M. & Koller, D. (2010). Efficiently selecting regions for scene understanding. In *Computer Vision and Pattern Recognition (CVPR)*, 3217–3224, IEEE. 42, 43

[58] Kung, H.T. (1986). Why systolic architectures? 300–309. 69

[59] l. Gaudiot, J., Bic, L., Dennis, J. & Dennis, J.B. (1994). Stream data types for signal processing. In *In Advances in Dataflow Architecture and Multithreading*, IEEE Computer Society Press. 66, 76

[60] Lazebnik, S., Schmid, C. & Ponce, J. (2006). Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Proc. of Computer Vision and Pattern Recognition*, 2169–2178, IEEE. 5, 57, 58, 64, 65

[61] LeCun, Y. & Bottou, L. (2002). Lush reference manual. Tech. rep., code available at http://lush.sourceforge.net. 76

[62] LeCun, Y. & Cortes, C. (1998). Mnist dataset. Http://yann.lecun.com/exdb/mnist/. 64

[63] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. & Jackel, L.D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*. 7, 8, 61

[64] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. & Jackel, L.D. (1990). Handwritten digit recognition with a back-propagation network. In *NIPS'89*. 5, 8, 30, 59

[65] LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, **86**, 2278–2324. 5, 8, 18, 25, 30, 50, 57, 59, 62, 80

[66] LeCun, Y., Bottou, L., Orr, G. & Muller, K. (1998). Efficient backprop. In G. Orr & M. K., eds., *Neural Networks: Tricks of the trade*, Springer. 31, 62

[67] LeCun, Y., Muller, U., Ben, J., Cosatto, E. & Flepp, B. (2005). Off-road obstacle avoidance through end-to-end learning. In *Advances in Neural Information Processing Systems (NIPS 2005)*, MIT Press. 8

[68] Lee, E.A. & David (1987). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, **36**, 24–35. 76

[69] Lee, H., Grosse, R., Ranganath, R. & Ng, Y., Andrew (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proc. of International Conference on Machine Learning (ICML'09)*. 51, 58, 62

[70] Lempitsky, V., Vedaldi, A. & Zisserman, A. (2011). A pylon model for semantic segmentation. In *Advances in Neural Information Processing Systems*. 42, 43, 47

[71] Lindholm, E., Nickolls, J., Oberman, S. & Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, **28**, 39–55. 21

[72] Liu, C., Yuen, J. & Torralba, A. (2009). Nonparametric scene parsing: Label transfer via dense scene alignment. *Artificial Intelligence*. 41, 42

[73] Lowe, D.G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*. 5, 57, 65

[74] Lyu, S. & Simoncelli, E.P. (2008). Nonlinear image representation using divisive normalization. In *CVPR*. 7, 61

[75] Mozer, M. (1991). *The Perception of Multiple Objects, A Connectionist Approach*. MIT Press. 9

[76] Munoz, D., Bagnell, J. & Hebert, M. (2010). Stacked hierarchical labeling. *ECCV 2010*. 42

[77] Mutch, J. & Lowe, D.G. (2006). Multiclass object recognition with sparse, localized features. In *CVPR*. 5, 9, 57

[78] Najman, L. & Schmitt, M. (1996). Geodesic saliency of watershed contours and hierarchical segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, **18**, 1163–1173. 38

[79] Najman, L., Cousty, J. & Perret, B. (2013). Playing with Kruskal: algorithms for morphological trees in edge-weighted graphs. In R.S. C.L. Luengo Hendriks G. Borgefors, ed., *International Symposium on Mathematical Morphology*, vol. 7883 of *Lecture Notes in Computer Science*, 135–146, Springer, Uppsala, Suède, kidico ANR-2010-BLAN-0205-03 and "Programme d'Investissements d'Avenir" (LabEx BEZOUT ANR-10-LABX-58). 15

[80] Nasse, F., Thurau, C. & Fink, G.A. (2009). Face detection using gpu-based convolutional neural networks. 8

[81] Ning, F., Delhomme, D., LeCun, Y., Piano, F., Bottou, L. & Barbano, P. (2005). Toward automatic phenotyping of developing embryos from

videos. *IEEE Trans. on Image Processing*, special issue on Molecular & Cellular Bioimaging. 9

[82] Nowlan, S. & Platt, J. (1995). A convolutional neural network hand tracker. In *Neural Information Processing Systems*, 901–908, Morgan Kaufmann, San Mateo, CA. 8

[83] Olshausen, B.A. & Field, D.J. (1997). Sparse coding with an overcomplete basis set: a strategy employed by v1? *Vision Research*. 62

[84] Osadchy, M., LeCun, Y. & Miller, M. (2007). Synergistic face detection and pose estimation with energy-based models. *Journal of Machine Learning Research*, **8**, 1197–1215. 8

[85] Park, C., Jung, J. & Ha, S. (2002). Extended synchronous dataflow for efficient dsp system prototyping. *Design Automation for Embedded Systems*, **6**, 295–322. 21

[86] Phi-Hung Pham, C.F.B.M.Y.L., Darko Jelaca & Culurciello, E. (2012). Neuflow: Dataflow vision processing system-on-a-chip. In *IEEE International Midwest Symposium on Circuits and systems*, Boise, Idaho, USA. 87

[87] Pinto, N., Cox, D.D. & DiCarlo, J.J. (2008). Why is real-world visual object recognition hard? *PLoS Comput Biol*, **4**, e27. 5, 7, 9, 57, 61

[88] Ranzato, M., Boureau, Y. & LeCun, Y. (2007). Sparse feature learning for deep belief networks. In *NIPS'07*. 9, 62, 64

[89] Ranzato, M., Huang, F., Boureau, Y. & LeCun, Y. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *Proc. of Computer Vision and Pattern Recognition*, IEEE. 51, 58

[90] Russell, B., Torralba, A., Liu, C., Fergus, R. & Freeman, W. (2007). Object recognition by scene alignment. In *Neural Advances in Neural Information*. 41

[91] Russell, B., Torralba, A., Murphy, K. & Freeman, W.T. (2007). Labelme: a database and web-based tool for image annotation. *International Journal of Computer Vision*. 78, 79

[92] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R. *et al.* (2008). Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, vol. 27, 18, ACM. 21

[93] Serre, T., Wolf, L. & Poggio, T. (2005). Object recognition with features inspired by visual cortex. In *CVPR*. 5, 9, 57, 58

[94] Shotton, J., Winn, J.M., Rother, C. & Criminisi, A. (2006). *TextonBoost*: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. In A. Leonardis, H. Bischof & A. Pinz, eds., *ECCV (1)*, vol. 3951 of *Lecture Notes in Computer Science*, 1–15, Springer. 35

[95] Simard, P., Steinkraus, D. & Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, vol. 2, 958–962. 45

[96] Simard, Y., Patrice, Steinkraus, D. & Platt, J.C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*. 7, 8, 61

[97] Socher, R., Lin, C.C., Ng, A.Y. & Manning, C.D. (2011). Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*. 42

[98] Tighe, J. & Lazebnik, S. (2010). Superparsing: scalable nonparametric image parsing with superpixels. *ECCV*, 352–365. 41, 42, 44

[99] Torralba, A. & Efros, A.A. (2011). Unbiased look at dataset bias. In *CVPR*, 1521–1528, IEEE. 47

[100] Turaga, S., Briggman, K., Helmstaedter, M., Denk, W. & Seung, H. (2009). Maximin affinity learning of image segmentation. *NIPS*. 17, 51

[101] Vaillant, R., Monrocq, C. & LeCun, Y. (1994). Original approach for the localisation of objects in images. *IEE Proc on Vision, Image, and Signal Processing*, **141**, 245–250. 8

[102] Vincent, P., Larochelle, H., Bengio, Y. & Manzagol, P.A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proc. 25th Annual International Conference on Machine Learning (ICML 2008)*, Omnipress. 10

[103] Weston, J., Rattle, F. & Collobert, R. (2008). Deep learning via semi-supervised embedding. In *ICML*. 62

Resume

Education

2010 - 2013 : PhD Student at *Université Paris Est.* Advised by Profs Laurent Najman and Yann LeCun (NYU), my PhD thesis focuses on Real-Time Image Understanding, from an algorithmic and computational point of view. Most of my thesis work was carried out at Pf Yann LeCun's lab at New York University.

2002 - 2008 : M.Eng in EE, received with honors at *Institut National des Sciences Appliquées.* Final year's major in Image and Signal Processing. Relevant courses: Computer Science, Control Theory, Electronics, Telecommunication, Mathematics (probability theory, optimization) and Physics.

Experience

2013 - now : Co-founder and CTO at *Madbits.* MadBits is a tech startup that focuses on building new user experiences in the media space (photos and videos). It builds on technologies I developed during my PhD.

2008 - 2013 : Research Scientist at the *Courant Institute, New York University.* 4 years of research on high-performance hardware for computer vision and convolutional networks. 3 years of research on image/scene parsing, based on deep-learning. These 5 years of work represent the core of my PhD thesis. I also did my Masters' thesis work at NYU, under the supervision of Prof. Yann LeCun.

2009 - 2011 : Visitor Scientist at *Yale University* Development of a fully custom dataflow processor—NeuFlow—for complex/generic vision

tasks. This work was done both at Yale's *e⁻Lab* (Prof. Eugenio Culurciello) and New York University (Prof. Yann LeCun). **Patent pending** – `http://neuflow.org`

2006 - 2007 : Junior Research Scientist at *University of New South Wales (ADFA)*. Development of a miniaturized FPGA based system to analyze video streams from a camera. This system was designed to be embedded in an unmanned helicopter (UAV) to provide the main computer with visual guidance.

2005 - 2008 : Private Teacher at *Acadomia*. Giving private lessons to students in mathematics and physics.

Journals

C. Farabet, C. Couprie, L. Najman and Y. LeCun, "Learning Hierarchical Features for Scene Labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, in press, 2013.

C. Farabet, R. Paz, J. Perez-Carrasco, C. Zamarreno, A. Linares-Barranco, Y. LeCun, E. Culurciello, T. Serrano-Gotarredona and B. Linares-Barranco, "Comparison Between Frame-Constrained Fix-Pixel-Value and Frame-Free Spiking-Dynamic-Pixel ConvNets for Visual Processing", in *Frontiers in Neuroscience*, 2012.

C. Farabet, Y. LeCun, K. Kavukcuoglu, B. Martini, P. Akselrod, S. Talay, and E. Culurciello, "Large-Scale FPGA-Based Convolutional Networks", in R. Bekkerman, M. Bilenko, and J. Langford (Ed.), *Scaling Up Machine Learning*, Cambridge University Press, 2011.

M. Garratt, H. Pota, A. Lambert, S. E.-Maslin and C. Farabet, "Visual Tracking and LIDAR Relative Positioning for Automated Launch and Recovery of an Unmanned Rotorcraft from Ships at Sea", in *Naval Engineers Journal*, vol 121, no. 2, pp. 99-110, June 2009.

International Conferences

C. Couprie, C. Farabet, L. Najman, Y. LeCun, "Indoor Semantic Segmentation using depth information", in *Proceedings of the International Conference on Learning Representations*, May 2013.

C. Culurciello, J. Bates, A. Dundar, J. Carrasco, C. Farabet, "Clustering Learning for Robotic Vision", ArXiv preprint, January 2013, in *Proceedings of the International Conference on Learning Representations*, May 2013.

C. Farabet, C. Couprie, L. Najman, Y. LeCun, "Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers", in *Proc. of the International Conference on Machine Learning (ICML'12)*, Edinburgh, Scotland, 2012. Video: `http://techtalks.tv/talks/57300/`

Phi-Hung Pham, Darko Jelaca, Clement Farabet, Berin Martini, Yann LeCun and Eugenio Culurciello, "NeuFlow: Dataflow Vision Processing System-on-a-Chip", in *IEEE International Midwest Symposium on Circuits and systems*, IEEE MWSCAS, 2012, Boise, Idaho, USA.

C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision", *in Proc. of the Fifth IEEE Workshop on Embedded Computer Vision (ECV'11 @ CVPR'11)*, IEEE, Colorado Springs, 2011. Invited Paper.

C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun and E. Culurciello, "Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems", in *International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, Paris, 2010.

Y. LeCun, K. Kavukcuoglu and C. Farabet, "Convolutional Networks and Applications in Vision", in *International Symposium on Circuits and Systems (ISCAS'10)*, IEEE, Paris, 2010.

C. Farabet, C. Poulet and Y. LeCun, "An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks", in *Proc. of the Fifth IEEE Workshop on Embedded Computer Vision (ECV'09 @ ICCV'09)*, IEEE, Kyoto, 2009.

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 (FPL'09)*, IEEE, Prague, 2009.

M. Garratt, H. Pota, A. Lambert, S. E.-Maslin and C. Farabet, "Visual Tracking and LIDAR Relative Positioning for Automated Launch and Recovery of an Unmanned Rotorcraft from Ships at Sea", in *ASNE Conference on Launch and Recovery of Manned and Unmanned Vehicles From Surface Platforms*, American Society of Engineers, Annapolis, 2008.

Workshops, Talks, Lectures, ...

C. Farabet, Invited talk at the Rowland Institute at Harvard (David Cox's lab), August 2012.

C. Farabet, J. Bergstra, Gave a series of lectures (tutorials) on deep-learning and feature learning, at the IPAM Graduate Summer School, July 2012.

C. Farabet, Invited talk at Gatsby, London, May 2012.

C. Farabet, Y. LeCun, Talk at the Big Learning Workshop, NIPS, 2011. Video: `http://www.youtube.com/watch?v=KaJtT1K3GtI`

C. Farabet, P. Akselrod, B. Martini, K. Kavukcuoglu, B. Corda, S. Talay, E. Culurciello and Y. LeCun, "A Dataflow Processor for General Purpose Vision", presented at *Neural Information Processing Systems (NIPS10)*, Vancouver, 2010.

B. Corda, C. Farabet, M. Scoffier and Y. LeCun, "Building Heterogeneous Platforms for End-to-end Online Learning Based on Dataflow Computing Design", in *Workshop on Learning on Cores, Clusters and Clouds (LCCC @ NIPS10)*, Whistler CA, 2010.

B. Corda, C. Farabet and Y. LeCun, "A Study of Parallel Computing for Machine Learning: Which Platform for Which Application", presented at the *4th Annual Machine Learning Symposium at the New York Academy of Sciences*, New York, 2010.

C. Farabet, "NeuFlow: a Vision Processor for Real-Time Object Categorization in Megapixel Videos", presented at *AIPR Workshop*, Washington DC, 2010.

C. Farabet, "NeuFlow: a Dataflow Computer for General Purpose Vision", presented at *e-labs Seminar Series*, Yale University, 2010.

C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun and E. Culurciello, "Bio-Inspired Processing for Ultra-Fast Object Categorization", in *High Performance Embedded Computing (HPEC10)*, MIT Lincoln Laboratory, Lexington, 2010.

C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "An FPGA-based Processor for Convolutional Networks", in *Snowbird Learning Workshop*, Clearwater FL, 2009.

C. Farabet, "Hardware Implementation of a Convolutional Neural Network – Design of a Neural Processor", Masters thesis, *INSA Lyon – New York University*, 2008.

C. Farabet, I. Ghizdavescu, S. Autin and C Revesz, "Cartography of a Wall in a Coke Oven by Reconstruction of a Panoramic Picture", *INSA Lyon*, 2008.

C. Farabet, "Implementation of a Tracking System within a FPGA", *University of New South Wales at ADFA*, 2007.

Software, Patent

R. Collobert, K. Kavukcuoglu, C. Farabet, "Torch7: A Matlab-like Environment for Machines Learning", in *Big Learning Workshop (@ NIPS'11)*, Sierra Nevada, Spain, 2011. `http://www.torch.ch`

C. Farabet, Y. LeCun, "Runtime Reconfigurable Dataflow Processor", filed on May 24, 2012. US Patent Application Number 13/479,742.

Reviewer

NIPS – Neural Information Processing Systems – `http://www.nips.cc`

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in identical or similar form to any other examination board.

The thesis work was conducted from 2008 to 2013 under the supervision of Prof. Yann LeCun, at the Courant Institute of Mathematical Sciences, New York University; and from 2010 to 2013 under the supervision of Prof. Laurent Najman, at Université Paris-Est.

New York,

Clément Farabet.