Large-Scale FPGA-based Convolutional Networks

Clément Farabet[1], Yann LeCun[1], Koray Kavukcuoglu[1],
Eugenio Culurciello[2], Berin Martini[2],
Polina Akselrod[2], Selcuk Talay[2]

1. The Courant Institute of Mathematical Sciences, New York University, New York, USA
2: Electrical Engineering Department, Yale University, New Haven, USA

Chapter in *Machine Learning on Very Large Data Sets*,
edited by Ron Bekkerman, Mikhail Bilenko, and John Langford,
Cambridge University Press, 2011.

May 2, 2011

Large-Scale FPGA-Based Convolutional Networks

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 (Lowe, 2004)) 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 (Lazebnik et al., 2006; Dalal and Triggs, 2005; Berg et al., 2005; Pinto et al., 2008).

Other models like HMAX-type models (Serre et al., 2005; Mutch and Lowe, 2006) 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 LeCun et al. (1998b) and Huang and LeCun (2006), pure supervised learning is used to update the parameters. However, recent works have focused on training with an auxiliary task (Ahmed et al., 2008) or using unsupervised objectives (Ranzato et al., 2007b; Kavukcuoglu et al., 2009; Jarrett et al., 2009; Lee et al., 2009).

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 (Lazebnik et al., 2006; Serre et al., 2005). 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 (Coates et al., 2009). 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.

1 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.

1.1 Convolutional Networks

convolutional network!overview

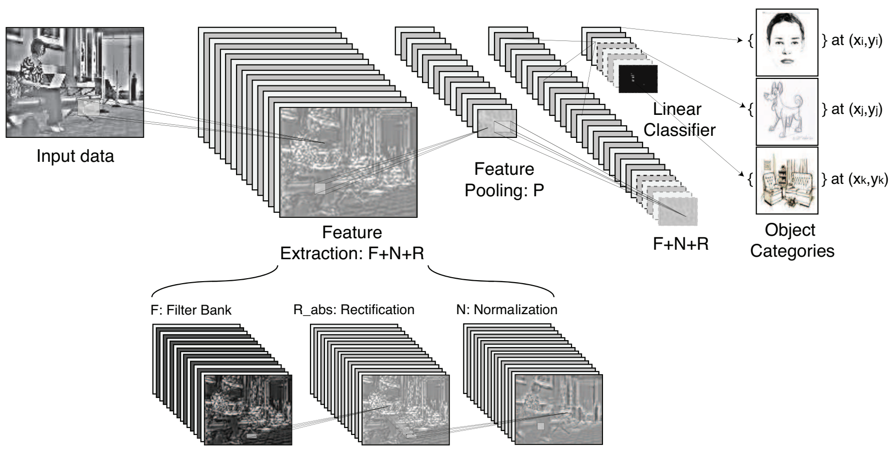

Figure 1: Architecture of a typical convolutional network for object recognition. This implements a convolutional feature extractor and a linear classifier for generic N-class object recognition. Once trained, the network can be computed on arbitrary large input images, producing a classification map as output.

Convolutional Networks (LeCun et al., 1990, 1998b) 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

$$y_j = b_j + \sum_i k_{ij} * x_i \qquad (1)$$

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

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

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:

$$v_{ijk} = x_{ijk} - \sum_{ipq} w_{pq}.x_{i,j+p,k+q}, \tag{3}$$

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

$$y_{ijk} = \frac{v_{ijk}}{\max(mean(\sigma_{jk}), \sigma_{jk})}, \tag{4}$$

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 (Lyu and Simoncelli, 2008; Pinto et al., 2008).

- **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 (LeCun et al., 1989; Simard et al., 2003). 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 (Kavukcuoglu et al., 2009).

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 LeCun et al. (1998b), and methods for efficient training are detailed in LeCun et al. (1998a).

1.2 History and Applications

convolutional network!history

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 (Fukushima and Miyake, 1982), 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 LeCun et al. (1989, 1990) 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, and led to several implementations, ranging from optical character recognition (OCR) to object detection, scene segmentation, and robot navigation:

- check reading (handwriting recognition) at AT&T (LeCun et al., 1998b) and Microsoft (Simard et al., 2003; Chellapilla et al., 2006),
- detection in images, including faces with record accuracy and real-time performance (Vaillant et al., 1994; Garcia and Delakis, 2004; Osadchy et al., 2007; Nasse et al., 2009), license plates and faces in Google's StreetView (Frome et al., 2009), or customers' gender and age at NEC,
- more experimental detection of hands/gestures (Nowlan and Platt, 1995), logos and text (Delakis and Garcia, 2008),
- vision-based navigation for off-road robots: in the DARPA-sponsored LAGR program, ConvNets were used for long-range obstacle detection (Hadsell et al., 2009). In Hadsell et al. (2009), the system is pre-trained off-line using a combination of unsupervised learning (as described in section 1.3) 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 (Jain and Seung, 2008) and image segmentation, particularly for biological images (Ning et al., 2005).

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 (Mozer, 1991), and the so-called HMAX family of models from T. Poggio's lab at MIT (Serre et al., 2005; Mutch and Lowe, 2006), 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. (Pinto et al., 2008) have identified the most appropriate non-linearities and normalizations by running systematic experiments with a single-stage architecture using GPU-based parallel hardware.

1.3 Unsupervised Learning of ConvNets

convolutional network!unsupervised learning

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 (Hinton and Salakhutdinov, 2006; Bengio et al., 2007; Ranzato et al., 2007a) 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 Jarrett et al. (2009), 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 Lee et al. (2009), 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* (Ahmed et al., 2008; Weston et al., 2008) helps regularizing the system and produces excellent performance.

1.3.1 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) (Kavukcuoglu et al., 2008). Similar to the well-known sparse coding algorithms (Olshausen and Field, 1997), inputs are approximated as a sparse linear combination of dictionary elements.

$$Z^* = \min_Z \|X - WZ\|_2^2 + \lambda |Z|_1 \tag{5}$$

In conventional sparse coding (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:

$$E(Z, W, K) = \|X - WZ\|_2^2 + \lambda \|Z\|_1 + \|Z - C(X, K)\|_2^2 \tag{6}$$

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 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).

1.3.2 Results on Object Recognition

convolutional network!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

Table 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 $[\mathbf{64.F^{9\times9} - R/N/P^{5\times5} - logreg}]$				
	$\mathbf{F - R_{abs} - N - P_A}$	$\mathbf{F - R_{abs} - P_A}$	$\mathbf{F - N - P_M}$	$\mathbf{F - P_A}$
$\mathbf{U^+}$	54.2%	50.0%	44.3%	14.5%
$\mathbf{X^+}$	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 $[\mathbf{256.F^{9\times9} - R/N/P^{4\times4} - logreg}]$				
	$\mathbf{F - R_{abs} - N - P_A}$	$\mathbf{F - R_{abs} - P_A}$	$\mathbf{F - N - P_M}$	$\mathbf{F - P_A}$
$\mathbf{U^+}$	65.5%	60.5%	61.0%	32.0%
$\mathbf{X^+}$	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%

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 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 (Lazebnik et al., 2006). 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 Jarrett et al. (2009).

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 (LeCun and Cortes, 1998): 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% (Ranzato et al., 2007a).

1.3.3 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 extraction systems like SIFT (Lowe, 2004), HoG (Dalal and Triggs, 2005) 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 (Lazebnik et al., 2006) classifier can also be seen as another layer of feature extraction since it performs a K-means based feature extraction followed by local histogramming.

2 A Dedicated Digital Hardware Architecture

convolutional network!hardware architecture FPGA ASIC

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 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 1.1, 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). 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. The size of internal memory caches growing according to Moore's Law, more data can fit internally, which naturally pulls performances 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.*

2.1 A Data-Flow Approach

data-flow computing

The data-flow hardware architecture was initiated by Adams (1969), and quickly became an active field of research (Dennis and Misunas, 1974; Hicks et al., 1993; l. Gaudiot et al., 1994). Cho et al. (2008) presents one of the latest data-flow architectures that has several similarities to the approach presented here.

Figure 2 shows a data-flow architecture whose goal is to process homogeneous streams of data in parallel (Farabet et al., 2010). 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. 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,

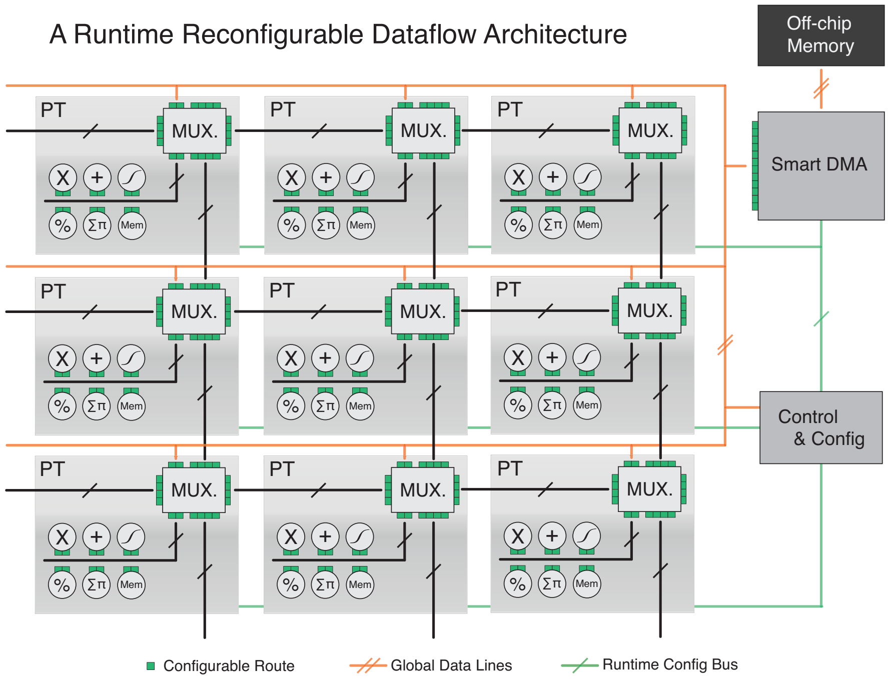

Figure 2: A data-flow computer. A set of runtime configurable processing tiles are connected on a 2D grid. They can exchange data with their 4 neighbors and with an off-chip memory via global lines.

- 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.2),
- a controller that can reconfigure most of the computing grid and the Smart DMA at runtime.

2.1.1 On Runtime Reconfiguration

reconfigurable hardware

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 (Dennis and Misunas, 1974; Hicks et al., 1993; Kung, 1986), 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 that 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 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 exit 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.

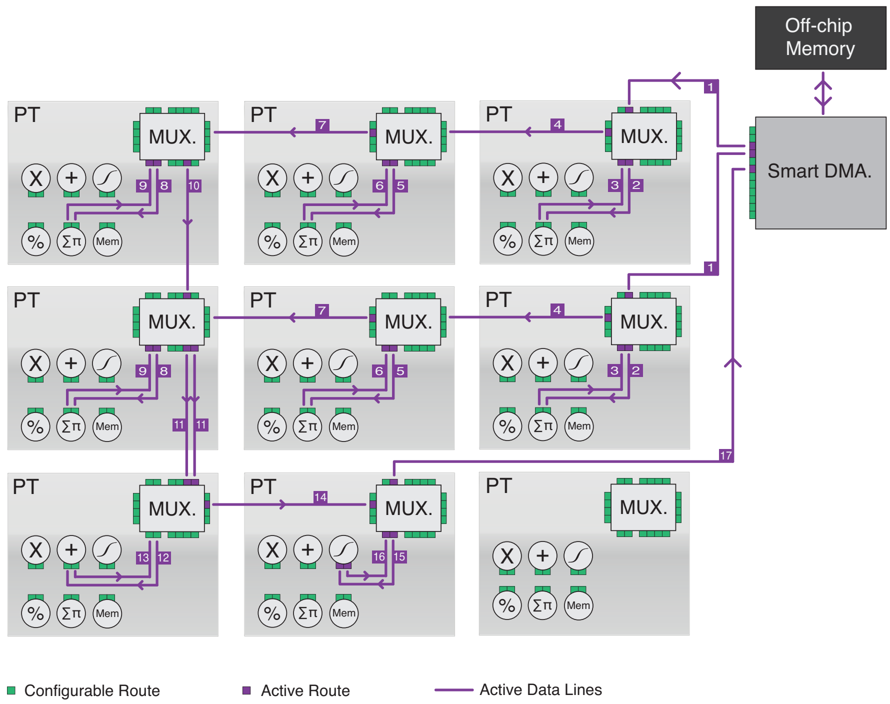

Figure 3: The grid is configured for a complex computation that involves several tiles: the 3 top tiles perform a 3×3 convolution, the 3 intermediate tiles another 3×3 convolution, the bottom left tile sums these two convolutions, and the bottom centre tile applies a function to the result.

Example 2. 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 shows an example of configuration, where the grid is configured to compute a sum of two convolutions followed by a non-linear activation function

$$y_{1,i,j} = Tanh(\sum_{m=0}^{K-1} \sum_{n=0}^{K-1} x_{1,i+m,j+n} w_{1,m,n} + \sum_{m=0}^{K-1} \sum_{n=0}^{K-1} x_{2,i+m,j+n} w_{2,m,n}). \qquad (7)$$

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 $B_{EXT}/3$.

2.2 An FPGA-Based ConvNet Processor

convolutional network!hardware implementation

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 2.1, 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 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.

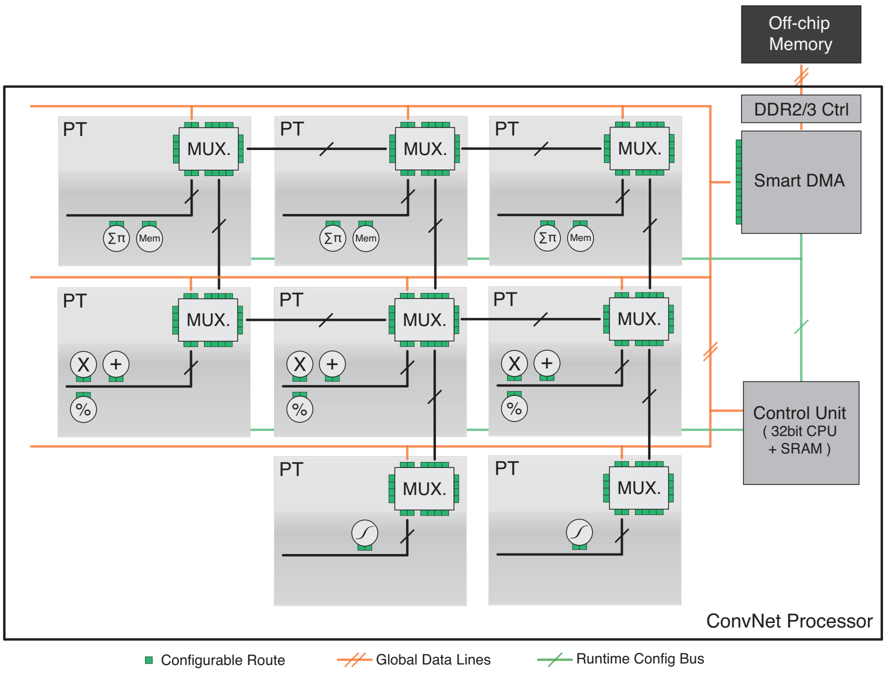

Figure 4: Overview of the ConvNet Processor system. A grid of multiple full-custom Processing Tiles tailored to ConvNet operations, and a fast streaming memory interface (Smart DMA).

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.

2.2.1 Specialized Processing Tiles

The *PTs* are independent processing tiles laid out on a two-dimensional grid. As presented in section 2.1, 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 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) (Farabet et al., 2009),
- 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 Farabet et al. (2009)), computes Equation 8 at every clock cycle.

$$y_{1,i,j} = x_{2,i,j} + \sum_{m=0}^{K-1}\sum_{n=0}^{K-1} x_{1,i+m,j+n} w_{1,m,n} \tag{8}$$

In equation 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.

A piece-wise linear decomposition is not as accurate (f is approximated by g, as in Eq. 9), but only requires a couple of coefficients a_i to represent a simple mapping such as a hyperbolic tangent, or a square root. 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. 10, the hardware can be reduced to shifters and adders only.

$$g(x) = a_i x + b_i \quad for \quad x \in [l_i, l_{i+1}] \tag{9}$$

$$a_i = \frac{1}{2^m} + \frac{1}{2^n} \quad m, n \in [0, 5]. \tag{10}$$

2.2.2 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 of 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}:

$$MAX(N_{DMA}) = \frac{B_{EXT}}{P_{bits}}. \tag{11}$$

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

2.3 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 (LeCun and Bottou, 2002) or Torch-5 (Collobert, 2008). Both libraries use the modular approach described in the introduction of section 2.

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 (Lee and David, 1987), and how to represent streams and computations on streams (l. Gaudiot et al., 1994). 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 3. 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 2, 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 2 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 4. 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 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:

$$y_j = \sum_{i=0}^{2} k_{ij} * x_i \quad for \quad j \in [0, 7]. \tag{12}$$

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 2 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.

2.4 Performance

Figure 5 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 possesses N_{KER} $K \times K$ learned kernels, N_{POOL} $P \times P$ learned pooling kernels, and N 200 dimension classification vectors.

Figure 5 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 2.1.1, 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).

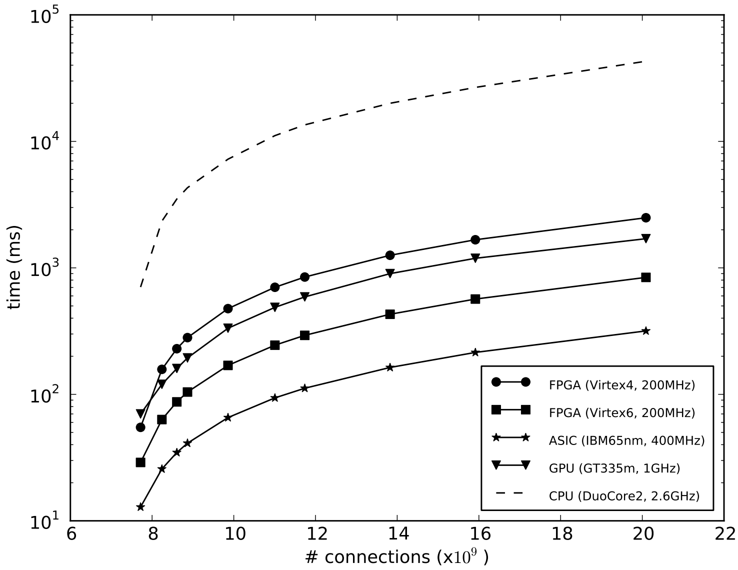

Figure 5: Compute time for a typical ConvNet (as seen in Figure 1).

2.4.1 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 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 6 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.

3 Summary

The convolutional network architecture is a remarkably versatile, yet conceptually simple paradigm that can be applied to a wide spectrum of perceptual tasks. While traditional ConvNets trained with supervised learning are very effective, training them requires a large number of labeled training samples. We have shown that using simple architectural tricks such as rectification and

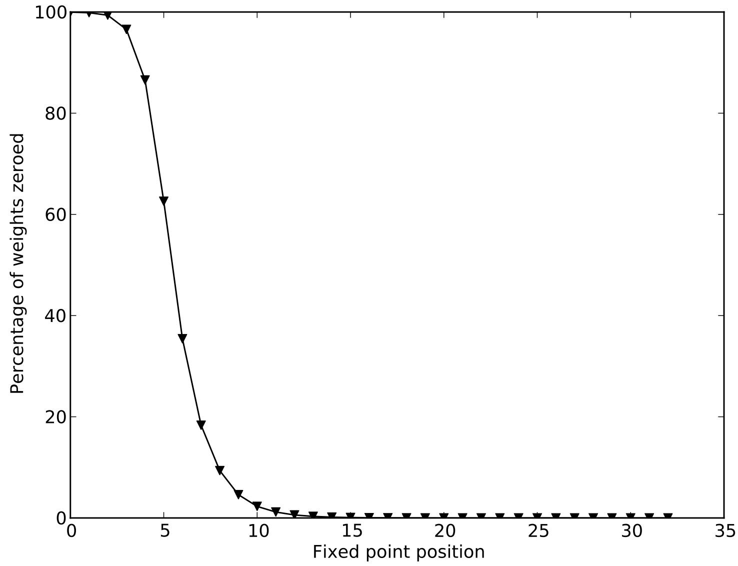

Figure 6: Quantization effect on trained networks: the x axis shows the fixed point position, the y axis the percentage of weights being zeroed after quantization.

contrast normalization, and using unsupervised pre-training of each filter bank, the need for labeled samples is considerably reduced.

We presented a data-flow computer that can be optimized to compute convolutional networks. Different use cases were studied, and it was seen that mapping/unrolling a convolutional network was straight-forward on such an architecture, thanks to their relatively uniform design.

Because of their applicability to a wide range of tasks, ConvNets are perfect candidates for hardware implementations, and embedded applications, as demonstrated by the increasing amount of work in this area. We expect to see many new embedded vision systems based on ConvNets in the next few years.

Future work on our data-flow architecture will aim at making it more general, to open the doors to more complex and generic recognition tasks. Multiple object detection (LeCun et al., 2004) or online learning for adaptive robot guidance (Hadsell et al., 2009) are tasks that will be largely improved by this system.

References

Adams, Duane Albert. 1969. *A computation model with data flow sequencing.* Ph.D. thesis, Stanford, CA, USA.

Ahmed, Amr, Yu, Kai, Xu, Wei, Gong, Yihong, and Xing, Eric. 2008. Training Hierarchical Feed-Forward Visual Recognition Models Using Transfer Learning from Pseudo-Tasks. In: *ECCV*. Springer-Verlag.

Bengio, Yoshua, Lamblin, Pascal, Popovici, Dan, and Larochelle, Hugo. 2007. Greedy Layer-Wise Training of Deep Networks. In: *NIPS*.

Berg, A. C., Berg, T. L., and Malik, J. 2005. Shape Matching and Object Recognition Using Low Distortion Correspondences. In: *CVPR*.

Chellapilla, Kumar, Shilman, Michael, and Simard, Patrice. 2006. Optimally combining a cascade of classifiers. In: *Proc. of Document Recognition and Retrieval 13, Electronic Imaging, 6067*.

Cho, Myong Hyon, chi Cheng, Chih, Kinsy, Michel, Suh, G. Edward, and Devadas, Srinivas. 2008. *Diastolic Arrays: Throughput-Driven Reconfigurable Computing*.

Coates, A., Baumstarck, P., Le, Q., and Ng, A.Y. 2009. Scalable learning for object detection with GPU hardware. Pages 4287–4293 of: *Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*. Citeseer.

Collobert, R. 2008. *Torch*. presented at the Workshop on Machine Learning Open Source Software, NIPS.

Dalal, Navneet, and Triggs, Bill. 2005. Histograms of Oriented Gradients for Human Detection. In: *CVPR*.

Delakis, M., and Garcia, C. 2008. Text Detection with Convolutional Neural Networks. In: *International Conference on Computer Vision Theory and Applications (VISAPP 2008)*.

Dennis, Jack B., and Misunas, David P. 1974. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, **3**(4), 126–132.

Farabet, Clément, Poulet, Cyril, Han, Jefferson Y., and LeCun, Yann. 2009. CNP: An FPGA-based Processor for Convolutional Networks. In: *International Conference on Field Programmable Logic and Applications (FPL'09)*. Prague: IEEE.

Farabet, Clément, Martini, Berin, Akselrod, Polina, Talay, Selcuk, LeCun, Yann, and Culurciello, Eugenio. 2010. Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems. In: *International Symposium on Circuits and Systems (ISCAS'10)*. Paris: IEEE.

Frome, A, Cheung, G, Abdulkader, A., Zennaro, M., Wu, B., Bissacco, A., Adam, H., Neven, H., and Vincent, L. 2009. Large-Scale Privacy Protection in Street-Level Imagery. In: *ICCV'09*.

Fukushima, Kunihiko, and Miyake, Sei. 1982. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, **15**(6), 455–469.

Garcia, C., and Delakis, M. 2004. Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Hadsell, Raia, Sermanet, Pierre, Scoffier, Marco, Erkan, Ayse, Kavackuoglu, Koray, Muller, Urs, and LeCun, Yann. 2009. Learning Long-Range Vision for Autonomous Off-Road Driving. *Journal of Field Robotics*, **26**(2), 120–144.

Hicks, James, Chiou, Derek, Ang, Boon Seong, and Arvind. 1993. *Performance Studies of Id on the Monsoon Dataflow System*.

Hinton, G E, and Salakhutdinov, R R. 2006. Reducing the dimensionality of data with neural networks. *Science*.

Huang, Fu-Jie, and LeCun, Yann. 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.

Jain, Viren, and Seung, H. Sebastian. 2008. Natural Image Denoising with Convolutional Networks. In: *Advances in Neural Information Processing Systems 21 (NIPS 2008)*. MIT Press.

Jarrett, Kevin, Kavukcuoglu, Koray, Ranzato, Marc'Aurelio, and LeCun, Yann. 2009. What is the Best Multi-Stage Architecture for Object Recognition? In: *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE.

Kavukcuoglu, Koray, Ranzato, Marc'Aurelio, and LeCun, Yann. 2008. *Fast Inference in Sparse Coding Algorithms with Applications to Object Recognition*. Tech. rept. Tech Report CBLL-TR-2008-12-01.

Kavukcuoglu, Koray, Ranzato, Marc'Aurelio, Fergus, Rob, and LeCun, Yann. 2009. Learning Invariant Features through Topographic Filter Maps. In: *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE.

Kung, H T. 1986. Why systolic architectures? 300–309.

l. Gaudiot, J., Bic, L., Dennis, Jack, and Dennis, Jack B. 1994. Stream Data Types for Signal Processing. In: *In Advances in Dataflow Architecture and Multithreading*. IEEE Computer Society Press.

Lazebnik, S., Schmid, C., and Ponce, J. 2006. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. Pages 2169–2178 of: *Proc. of Computer Vision and Pattern Recognition*. IEEE.

LeCun, Y., and Bottou, L. 2002. *Lush Reference Manual*. Tech. rept. code available at http://lush.sourceforge.net.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. 1990. Handwritten digit recognition with a back-propagation network. In: *NIPS'89*.

LeCun, Y., Bottou, L., Orr, G., and Muller, K. 1998a. Efficient BackProp. In: Orr, G., and K., Muller (eds), *Neural Networks: Tricks of the trade*. Springer.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. 1998b. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, **86**(11), 2278–2324.

LeCun, Yann, and Cortes, Corinna. 1998. *MNIST dataset*. http://yann.lecun.com/exdb/mnist/.

LeCun, Yann, Huang, Fu-Jie, and Bottou, Leon. 2004. Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In: *Proceedings of CVPR'04*. IEEE Press.

Lee, Edward Ashford, and David. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, **36**, 24–35.

Lee, Honglak, Grosse, Roger, Ranganath, Rajesh, and Ng, Andrew, Y. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: *Proc. of the 26th International Conference on Machine Learning (ICML'09)*.

Lowe, David G. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*.

Lyu, S, and Simoncelli, E P. 2008. Nonlinear image representation using divisive normalization. In: *CVPR*.

Mozer, M.C. 1991. *The Perception of Multiple Objects, A Connectionist Approach*. MIT Press.

Mutch, Jim, and Lowe, David G. 2006. Multiclass Object Recognition with Sparse, Localized Features. In: *CVPR*.

Nasse, Fabian, Thurau, Christian, and Fink, Gernot A. 2009. *Face detection using GPU-based convolutional neural networks*.

Ning, Feng, Delhomme, Damien, LeCun, Yann, Piano, Fabio, Bottou, Leon, and Barbano, Paolo. 2005. Toward Automatic Phenotyping of Developing Embryos from Videos. *IEEE Transactions on Image Processing*. Special issue on Molecular and Cellular Bioimaging.

Nowlan, S., and Platt, J. 1995. A Convolutional Neural Network Hand Tracker. Pages 901–908 of: *Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufmann.

Olshausen, B A, and Field, D J. 1997. Sparse coding with an overcomplete basis set: a strategy employed by V1? *Vision Research*.

Osadchy, M., LeCun, Y., and Miller, M. 2007. Synergistic Face Detection and Pose Estimation with Energy-Based Models. *Journal of Machine Learning Research*, **8**(May), 1197–1215.

Pinto, Nicolas, Cox, David D, and DiCarlo, James J. 2008. Why is Real-World Visual Object Recognition Hard? *PLoS Comput Biol*, **4**(1), e27.

Ranzato, Marc'Aurelio, Boureau, Y-Lan, and LeCun, Yann. 2007a. Sparse feature learning for deep belief networks. In: *NIPS'07*.

Ranzato, Marc'Aurelio, Huang, Fu-Jie, Boureau, Y-Lan, and LeCun, Yann. 2007b. Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition. In: *Proc. Computer Vision and Pattern Recognition Conference (CVPR'07)*. IEEE Press.

Serre, Thomas, Wolf, Lior, and Poggio, Tomaso. 2005. Object Recognition with Features Inspired by Visual Cortex. In: *CVPR*.

Simard, Patrice, Y., Steinkraus, Dave, and Platt, John C. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In: *ICDAR*.

Vaillant, R., Monrocq, C., and LeCun, Y. 1994. Original approach for the localisation of objects in images. *IEE Proc on Vision, Image, and Signal Processing*, **141**(4), 245–250.

Weston, J., Rattle, F., and Collobert, R. 2008. Deep Learning via Semi-Supervised Embedding. In: *ICML*.