DSP Design With Hardware Accelerator For Convolutional Neural Networks

JULIAN HILLE
LUCAS FERREIRA
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

DSP Design With Hardware Accelerator For Convolutional Neural Networks

Julian Hille
ju6441hi-s@student.lu.se
Lucas Ferreira
frt14lsa@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor:
Joachim Rodrigues
joachim.rodrigues@eit.lth.se
Hemanth Prabhu (Xernergic AB)
hemanth.prabhu@xenergic.com

Examiner:
Pietro Andreani
pietro.andreani@eit.lth.se

March 15, 2019

Printed in Sweden
Tryckeriet i E-huset, Lund

Acknowledgments

We would like to thank our thesis supervisors Associate Professor Joachim Rodrigues (Lund University) and to Hemanth Prabhu (Xenergic AB) for the support received during this thesis. We would like to thank you for the great opportunity at Xenergic of applying our knowledge regarding Embedded Electronic Engineering with the focus on the hardware-software co-design applied to machine learning applications. Finally we would like to show our gratitude to our families and friends for all the support and the opportunity to study in Sweden at Lund University. Thank you.

Lucas Ferreira and Julian Hille
March 2019

Abstract

Convolutional Neural Networks impressed the world in 2012 by reaching state-of-the-art accuracy levels in the ImageNet Large Scale Visual Recognition Challenge. The era of machine learning has arrived and with it countless applications varying from autonomous driving to unstructured robotic manipulation. Computational complexity in the past years has grown exponentially, requiring highly efficient low power new hardware architectures, capable of executing those. In this work, we have performed optimization in three levels of hardware design: from algorithmic, to system, and accelerator level. The design of a DSP with Tensilica and the integration of Xenergic dual port SRAMs, for direct memory access of a convolution hardware accelerator, lead to four orders speed-up on the initial identified bottleneck, causing an estimated three times final speed-up of a single handwritten classification image compared to the pure software implementation. Higher speed-up is expected for deeper convolutional architectures and larger image dimensions, due to the linear time complexity scaling of the convolution hardware accelerator in comparison to conventional non-linear software-based approaches.

Popular Science Summary

Artificial Intelligence is becoming more and more used in newer technologies, from mobile phones featuring voice detection to autonomous driving cars and also in the new industries. For such applications the "intelligence" requirements are increasing. Today much computations are solved by using the cloud. For example, in mobile phones, voice assistance only works with an internet connection. The same approaches are not possible for autonomic controlled vehicles. The essential control features have to be inside the vehicle.

Therefore we are in need to bring Artificial Intelligence into mobile devices. This thesis aims to implement a benchmark classification problem (MNIST) by using a programmable processor, designed with a commercial tool, and a flexible hardware accelerator to speed up a Convolutional Neural Network that recognizes handwritten digits between 0 and 9. Therefore we have designed and trained a reference architecture in the programming language Python, from which the weights were obtained to implement the same architecture on the designed processor (by using C/C++). By investigation of the most resources consuming functions, we have figured out that the convolution has the highest computation cost. Hence the accelerator was implemented and instructions added, directly connecting it to the processor. Results obtained achieved four orders of magnitude total speed-up of the identified bottleneck. Yielding in an estimated three times final speed-up for a single handwritten classification image, compared to a pure software implementation at the same processor. Additionally, an open-source processor alternative is proposed.

List of Acronyms and Abbreviations

AI Artificial Intelligence
ALU Arithmetic Logic Unit
ANN Artificial Neural Network
APB Advanced Peripheral Bus
DSP Digital Signal Processor
AXI Advanced Extensible Interface
CNN Convolutional Neural Network
FCU Fast FIR Computation Unit
FIR Finite Impulse Response
FLIX Flexible Length Instruction Extensions
GPIO General-Purpose Input/Output
ISA Instruction Set Architecture
ML Machine Learning
MLP Multi-Layer-Perceptron
MNIST Modified National Institute of Standards and Technology
PCA Principal Component Analysis
PIF Processor Interface
RISC Reduced Instruction Set Computer
RTL Register Transfer Level
SIMD Single Instruction, Multiple Data
SRAM Static Random-Access Memory
TIE Tensilica Instruction Extension
T-SNE T-Distributed Stochastic Neighbor Embedding
VLIW Very Long Instruction Word
XPG Xtensa Processor Generator

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As of psychological definitions of intelligence, "ability of an organism to solve new problems" [1] or "the capacity to learn or to profit by experience"[2], Artificial Intelligence (AI) aims to create entities capable of adapting itself to (out)perform in human-like activities.

AI can be categorized into three different stages: Artificial Narrow Intelligence (ANI), Artificial General Intelligence (AGI), and Artificial Super Intelligence (ASI). Cutting edge AI systems currently are in the first stage (ANI), executing application specific tasks without self-expanding abilities, even outperforming human skills in such repetitive tasks, for instance, driving, GO (game) [3], and skin cancer diagnosis [4]. At the AGI stage, also known as the singularity, machines will be able to perform a broad range of tasks, being able of competing with any human intellectual activity, consequently making the mental distinction between people and machine unclear. Finally, at the ASI stage, which shall occur shortly after the singularity, those entities will cognitively outperform the most gifted human level in most of the efforts.

AI can be divided into two broad categories: unsupervised and supervised learning [5]. The more human-like intelligence is unsupervised learning, where a computer is not given any labeled data or guidance, and it has to find substructures or rules in the given data to perform a task. It can be applied in a broad range of functions, including clustering and anomaly detection problems (identification of unexpected patterns). Clustering involves any grouping problems, for instance, defining personalities by interests, creating profiles based on people's activity monitoring, grouping similar images, or audios and *etc.*.

The most common form of machine learning is the supervised learning, where the computer receives input data and expected output (*e.g.*, label) or feedback for the training. One of the classic examples of the supervised learning algorithm is the cat *vs.* dog recognition. Imagine that we have a dataset containing thousands of labeled images of cats and dogs, and we train a neural network (set of neurons, implementing a regression) to guess whether the picture is a dog or a cat. Based on the predicted output, we compare it to the image's label, and we compute a loss function which will update our prediction model.

Reinforcement learning is a particular case within supervised learning, where only rewards or punishments are given to the system as feedback of its actions. This methodology gained status when Google's *AlphaGo* overcame the world champion go player for three times in 2017 [6], or when *OpenAI* Bot defeated consistently

professional players of the intricate game Dota 2 [7]. Not only for games this methodology exhibited its potential, but also by observing results in extensive fields such as finance, energy distribution, medicine, and robotics, in which for example a robotic hand was designed to manipulate physical objects exhibiting unprecedented dexterity [8].

The recent AI boom dates back mainly from results obtained by using a deep convolutional network algorithm (a subset of supervised learning) developed by [9], applied to artificial vision in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. This annual contest is based on ImageNet dataset, which consists of over 15 million labeled images, distributed over 22,000 different categories. ILSVRC uses a subset of ImageNet with roughly 1,000 images distributed over 1,200 categories, and data is divided into 1.2 million images for training, 50,000 for validation and 150,000 for testing. Their proposed architecture resulted in top-1 and top-5 error rates (error rate of which the correct label is not one of the five most probable predictions by the model) of 37.5% and 17.0%, which became 2012's state-of-the-art for the given dataset.

Convolutional Neural Networks (CNNs) became then mainly applied in machine learning for computer vision. Its underlying architecture contains three different operations to predict the output: layers of convolution filters are applied to the input, having pooling and activation operations in between. The outputs after each activation, also known as activation maps (or feature maps), are connected to the next convolution layer. The last one serves as an input for a fully connected neural network, which then predicts the output, followed by a classification layer performed primarily by the linear or softmax classifiers.

CNNs, as we know, were firstly introduced by LeCun (1989) [10] to identify handwritten zip codes in the US, together with very similar approaches in the same year as the Time Delay Neural Networks, or TDNNs to identify acoustic-phonetics [11]. All those based on [12], and his ideas inspired on the cortex simple and complex cells corresponding to small regions of the visual field. Although those ideas date back from the '80s, not much progress was made until the previously referred works in 2012. Reasons are that the amount of calculations performed by a standard CNN scales fast, depending on the number of layers and input size, requiring substantial greater complexity regarding hardware (specifically memories and processing elements).

Another limitation was the amount of data required to train the neural network, since it requires vast amounts of structured labeled data, demanding reasonable efforts. For these reasons, CNNs did not perform astonishingly well, and was a topic treated with skepticism by most in the area of machine learning until recent results. To make it worse, back propagation was considered risky, as it was thought that chances were high of the algorithm getting trapped in a local minimum, hypothesis denied by [13]. Finally, 20 years after LeCun [10] firstly introduced CNN, the era of data and more capable hardware has arrived, and with it a new spring for the CNNs.

In this thesis we are using the knowledge about CNNs to analyze the bottleneck and to develop a hardware accelerator to increase efficiency of the forward-propagation computation. First of all, we explore into the design of a Digital Signal Processor (DSP) with Cadence Tensilica (a tool to design and generate a processor

for specific applications) by adding our own instructions. Afterwards, we extend the processor with the hardware accelerator. Therefore this thesis begins with background information about Artificial Neural Networks (ANNs) and CNNs by going through a deeper understanding of the networks. Afterwards, we design and train our CNN architecture to solve a given task performing the most power demanding CNNs function in special-purpose designed hardware accelerator. Subsequently, we describe the possibilities in Tensilica and an open-source alternative by usage of the RISC V. A proposed extension to the core containing the aforementioned accelerator and Xenergic's dual port Static Random-Access Memorys (SRAMs) follows. At the end we discuss future projects and summarize main ideas.

Chapter 2

Feed-forward Networks and Convolution Neural Networks

2.1 Feed-forward Neural Networks

2.1.1 Perceptron

In the following section we are using the single perceptron as illustrated in Figure 2.1 to explain the simplest ANN, gaining a deeper understanding of CNNs building blocks based on [14].

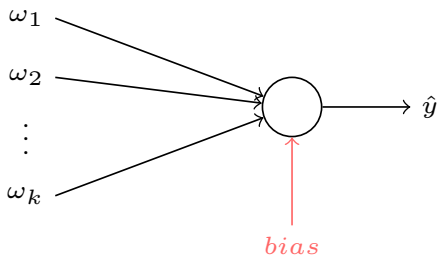


Figure 2.1: Diagram of a single perceptron.

The perceptron's outputs, for N experiments, can be calculated as:

$$\hat{\mathbf{y}} = \sigma\left(\mathbf{w}^T\mathbf{X} + b\right) = \sigma\left(\mathbf{w}^T\mathbf{X}\right) \ , \tag{2.1}$$

where $\mathbf{w}$ is a weight vector of dimension (M+1)x1 of the perceptron, $\mathbf{X}$ is an input matrix of dimension (M+1)xN, b is the bias value, and σ an activation function. Additionally, as shown, the bias term can be added as an extra weight ω_0 into the perceptron.

The weight vector $\mathbf{w}$ is given by:

$$\mathbf{w}^T = [\omega_0 \quad \omega_1 \quad \omega_2 \quad \dots \quad \omega_N] \ , \tag{2.2}$$

and $\mathbf{X}$ is the data input matrix for N experiments having the form:

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ x_{11} & x_{11} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \dots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{pmatrix} . \tag{2.3}$$

Even though the simplest ANN model, the perceptron is powerful enough to solve any hyperplane separable classification problem. Let us consider for simplicity the two-dimensional case, where two different classes are learned by the single percetron.

For this two-dimensional classification (Figure 2.2), if separable, there will be a decision boundary which fully separates the two classes. The decision line is the hyperplane learned by the perceptron. Any element below this line belongs to class X1, and consequently the others to class X2.

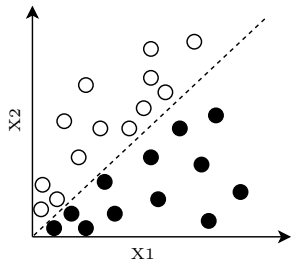


Figure 2.2: Linear classification of a two-class problem learned by a single perceptron.

The learning algorithm can be seen as a method which updates the weights, and consequently moving the decision plane such that the two classes can be fully separated. The method is based on calculating the output for each experiment and comparing the estimated output with the real result. In case of error between estimated and expected, then the new weights, *i.e.* $\mathbf{w}\,(t+1)$, are updated following the rule:

$$\mathbf{w}\,(t+1) = \mathbf{w}\,(t) + \eta\,(\hat{y}|\mathbf{x}_\mu - y|\mathbf{x}_\mu)\,\mathbf{x}_\mu\ , \tag{2.4}$$

where $\hat{y}|\mathbf{x}_\mu$ represents the predicted output given a single experiment $\mathbf{x}_\mu$, and $y|\mathbf{x}_\mu$ the expected output. The factor η, also called the learning rate, expresses the relevance of a single experiment error in comparison to all other previous updates also being a compromise between training speed and final accuracy.

A classification problem is said linear separable if there is a hyperplane which can separate the two classes. As a consequence, (N)AND, (N)OR, or some other variations, can be successfully realized by a perceptron. For instance, let us consider the following logic:

$$\overline{X2\overline{X1}} = \overline{X2} + X1\ \ .$$

Illustrated by Figure 2.3 has the following truth table:

X2	X1	Y
-1	-1	1
-1	1	1
1	-1	-1
1	1	1

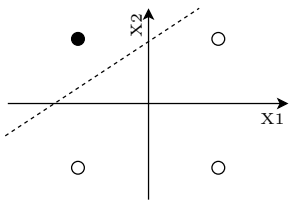


Figure 2.3: Example of complexity of linear separable logic, and addressed by a single perceptron.

Such logic can be implemented by a single perceptron with weights equals to:

$$\mathbf{w}^T = [\omega_0 \quad \omega_1 \quad \omega_2] = [1 \quad 1 \quad 1] \tag{2.5}$$

and, for instance, the sign function serving as activation defined as:

$$\sigma = \text{sgn} = \begin{cases} 1, x > 0 \\ -1, x < 0 \, . \end{cases} \tag{2.6}$$

We can easily verify that:

$$\begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix} = \sigma \left[\begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}^T \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{pmatrix} \right] \tag{2.7}$$

fully solves the truth table, hence it is a possible solution learned by single perceptron to solve the $\overline{X2} + X1$ problem.

Direct Solution for the weights

A learning algorithm, for N experiments, can be introduced considering first the linear case, *i.e.* when the activation $\sigma(x) = x$. Therefore Equation 2.1 turns into:

$$\hat{\mathbf{y}} = \sigma\left(\mathbf{w}^T \mathbf{X}\right) = \mathbf{w}^T \mathbf{X} \, . \tag{2.8}$$

An error function can be then defined as:

$$\begin{aligned} E(w) &= \sum_{n=1}^{N} \left(y_n - \hat{y}(x_n)\right)^2 \\ &= (\mathbf{y} - \hat{\mathbf{y}})^T (\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\hat{\mathbf{y}} - \hat{\mathbf{y}}^T\mathbf{y} + \hat{\mathbf{y}}^T\hat{\mathbf{y}} \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{y}^T(\mathbf{X}\mathbf{w}) - (\mathbf{X}\mathbf{w})^T\mathbf{y} + (\mathbf{X}\mathbf{w})^T(\mathbf{X}\mathbf{w}) \\ &= \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{w}^\mathbf{T}\mathbf{X}^\mathbf{T}\mathbf{X}\mathbf{w} \, . \end{aligned} \tag{2.9}$$

The learning algorithm obtains the weights $\mathbf{w}$ such that the error function is minimized. The weights which minimize the error are obtained when $\nabla_{\mathbf{w}} E = 0$, which represents the fact that a minimum can only occur when the derivative is zero. Thus:

$$\begin{aligned} \nabla_{\mathbf{w}} E &= -\mathbf{y}^T\mathbf{X} - \mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\mathbf{w} \\ &= 2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{X}^T\mathbf{y} = 0 \\ 2\mathbf{X}^T\mathbf{X}\mathbf{w} &= 2\mathbf{X}^T\mathbf{y} \\ \mathbf{w} &= \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y} \,, \end{aligned} \tag{2.10}$$

where $\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$ is also known as the pseudo inverse $\mathbf{X}^\dagger$. The average products over N experiments is also the correlation matrix of the inputs $\mathbf{R_{xx}} = \frac{1}{N}\mathbf{X}^T\mathbf{X}$, and finally $\mathbf{r_{xd}} = \frac{1}{N}\mathbf{X}^T\mathbf{y}$ the correlation between input and output. Thus 2.10 can also be written as:

$$\mathbf{w} = \mathbf{R_{xx}}^{-1}\mathbf{r_{xd}} \,, \tag{2.11}$$

which is also known as linear regression, also a possible feature implemented by the perceptron.

The direct learning algorithm can be seen merely as a closed-form solution of the linear system of equations involved with linear regression problems (Equation 2.10), and much more straightforward than any other learning alternatives. However, mostly used activation functions, capable of solving more complex tasks (linear regression), will result in nonlinear systems when trying to obtain the weights similarly as of minimizing an error function.

In case of non-linear equation systems, having a direct solution is rather rare. Thus another more appropriate learning algorithm is necessary to address more complex problems.

Gradient Descent

The gradient descent algorithm not only enables us to find out the weights for a broad range of problems, but also features an iterative method for updating the weights given multiple experiments.

The gradient descent method can also be calculated for the linear activation function. For convenience, the error function adopted for this derivation will be the same (as in Equation 2.9), but averaged over $1/2N$, resulting in

$$\begin{aligned} E\left(w\right) &= \frac{1}{2N}\sum_{n=1}^{N}\left(y_n - \hat{y}\left(x_n\right)\right)^2 \\ &= \frac{1}{2N}[\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\hat{\mathbf{y}} - \hat{\mathbf{y}}^T\mathbf{y} + \hat{\mathbf{y}}^T\hat{\mathbf{y}}] \\ &= \frac{1}{2N}[\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\mathbf{w} + (\mathbf{X}\mathbf{w})^T\mathbf{X}\mathbf{w}] \\ \frac{\partial E\left(w\right)}{\partial w_k} &= \frac{1}{2N}[-2\mathbf{y}^T\mathbf{X} + 2(\mathbf{X})^T\mathbf{X}\mathbf{w}] \\ &= \frac{1}{N}[\mathbf{X}^T\left(\mathbf{X}\mathbf{w} - \mathbf{y}\right)] \,. \end{aligned} \tag{2.12}$$

Thus, the weight update is given by a learning rate times the derivative of the error in respect to the weights, as:

$$\Delta \mathbf{w_k} = -\eta \frac{\partial E\left(w\right)}{\partial w_k} \ , \tag{2.13}$$

then the updates are:

$$\begin{aligned} \Delta \mathbf{w_k} &= \frac{\eta}{N}[\mathbf{X}^T\left(\mathbf{X}\mathbf{w} - \mathbf{y}\right)] \\ \Delta w_k &= \frac{\eta}{N}\sum_n \left(y - \hat{y}\right) x_n \\ &= \frac{\eta}{N}\sum_n \delta_n x_n \ , \end{aligned} \tag{2.14}$$

consequently, for online update (where the weights are updated after each input set), the update rule resumes in:

$$\Delta w_k = \eta \delta_n x_n \ . \tag{2.15}$$

It is also straight-forward to prove that in case of non-linear activation functions the gradient descent has the following weight update rule, when the activation function is differentiable, thus:

$$\begin{aligned} \Delta \mathbf{w_k} &= \frac{\eta}{N}[\mathbf{X}^T\left(\mathbf{X}\mathbf{w} - \mathbf{y}\right)\sigma\left(\alpha_n\right)^{'}] \\ \Delta w_k &= \frac{\eta}{N}\sum_n \left(y - \hat{y}\right) x_n \sigma\left(\alpha_n\right)^{'} \\ &= \frac{\eta}{N}\sum_n \delta_n x_n \ . \end{aligned} \tag{2.16}$$

2.1.2 Multi-Layer Perceptron

Even though very powerful for its simplicity, the perceptron cannot solve an exclusive-or (XOR) problem. XOR implements the following logic and truth table.

$$X1 \oplus X2 = \overline{X1}X2 + X1\overline{X2}$$

X2	X1	Y
-1	-1	-1
-1	1	1
1	-1	1
1	1	-1

Figure 2.4 illustrates that there is no possible decision plane capable of reaching linear separability for the XOR problem. Thus, it turns out that a composition of perceptrons is necessary for solving this problem.

The Multi-Layer-Perceptron (MLP) represented in Figure 2.5, is able to solve the XOR problem. A MLP is a composition of perceptrons, arranged in one or more

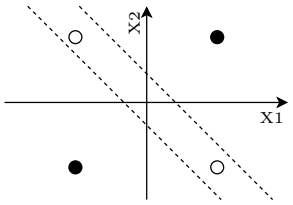


Figure 2.4: XOR classification with decision plans.

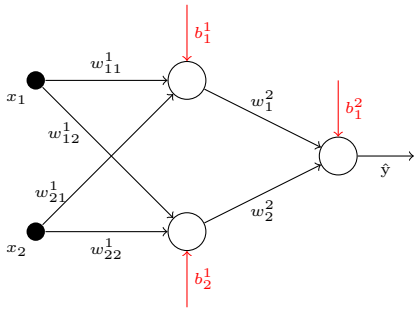


Figure 2.5: MLP capable of solving the XOR problem.

hidden-layers, whose activations are in most cases non-linear functions (commonly used are tanh, logistic, or rectifier). Also, an often used output activations function is softmax and used to represent the probability distribution over n different classes. Those are defined at Equations 2.17.

$$\begin{aligned} \sigma(x) &= \frac{1}{1+e^{-x}} \\ \tanh(x) &= \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \\ \mathrm{ReLu}(x) &= \max(0,x) \\ \mathrm{softmax}(x)_i &= \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}} \end{aligned} \tag{2.17}$$

The output of the MLP can be calculated as:

$$\begin{aligned} \hat{Y} &= \mathrm{sgn}\left(w_1^2 h_1 + w_2^2 h_2 + b_1^2\right) \\ h_1 &= \mathrm{sgn}\left(w_{11}^1 x_1 + w_{21}^1 x_2 + b_1^1\right) \\ h_2 &= \mathrm{sgn}\left(w_{12}^1 x_1 + w_{22}^1 x_2 + b_2^1\right) \ . \end{aligned} \tag{2.18}$$

resulting in the following outputs:

x2	x1	h1	h2	$\hat{Y}$
-1	-1	-1	-1	1
-1	1	1	-1	-1
1	-1	1	-1	-1
1	1	1	1	1

and consequently solving the XOR classification problem, since when the inputs are different the MLP outputs -1 and 1 in the opposite.

Deeper Understanding of the MLP

Undoubtedly, the multilayer perceptron is one of the essential building blocks of machine learning. It extends all the results shown for the single perceptron, and adds the feature of classifying any number of different classes, given a large enough MLP model, according to the universal approximation theorem [14].

The multilayer perceptron mechanics can be understood as a composition of perceptrons trying to fit a hyperplane capable of linearly separate a multi-class dataset, by performing affine transformations followed by nonlinear distortions of the dataspace.

For instance, let us consider first the one-dimensional classification problem:

Figure 2.6: Example of MLP 1-D mechanism.

An MLP capable of solving such a problem will first extend this 1-D problem into 2-D data space. Then a set of affine transformations (rotation and translations implemented as in Equation 2.1 by the weight matrix $\mathbf{w}$ and the bias respectively) followed by non-linear distortions (implemented by the activation function) will result in the following effect:

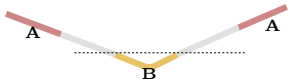


Figure 2.7: Example of a linearly separable 1-dimension problem solved with an MLP.

where the problem is linearly separable. This fundamental mechanism can still be visualized for a 2D case such as the spiral classification, illustrated in Figure 2.8. It is observable that the learning process, realized by training of the MLP, finds out the weights necessary to transform the given data into a linearly separable problem.

Linear separability, however, is not a simple task to ask from an MLP. In some cases where there is a mathematical knot in the dataset, going only one dimension higher (*i.e.* $n+1$) than the problem is not enough to guarantee linear separability. Such problems may require an MLP going up to $2n+2$, where any n-order manifold (links and knots are 1-dimensional manifolds) can be untangled. [15]

Such topological results are interesting to us because it forms lower boundaries of which the MLP cannot be smaller, in order to entirely solve the classification problem. For instance, a dataset as shown in Figure 2.9 is impossible to be solved by an MLP which does not have at least one hidden layer containing three (or more) neurons, no matter how deep the architecture. [15]

Now that we have a deeper understanding of what is asked from an MLP, and its mechanism to classify different data, it is time to update the main results from the

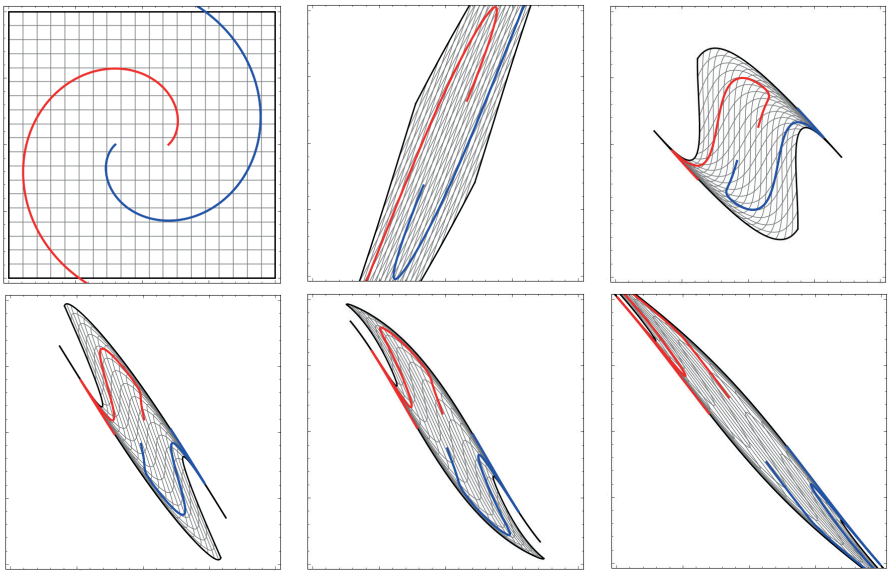


Figure 2.8: Effect of forward propagation on different stages for a 2D spiral classification problem.[15]

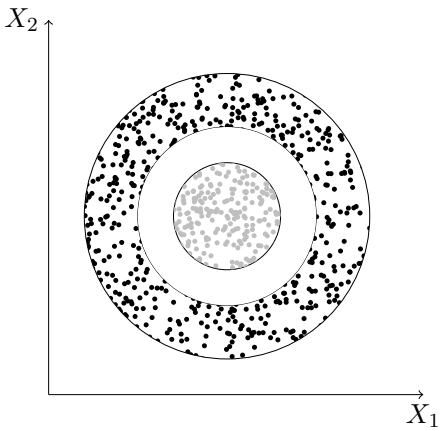


Figure 2.9: Dataset requires at least three neurons in one of the hidden layers MLP, for linear separability.

single perceptron (forward- and back-propagation algorithms). This time, however, such expressions will not be derived, since the basic strategies, as derived for the perceptron, still holds true.

Figure 2.10 serves as an example of an MLP. Observe that the number of neurons at each of the layers, and the actual depth (*i.e.* how many hidden layers are utilized) is defined by the user. Research as in [16] shows extensively how deeper models positively impact final accuracy, however as discussed before, depth alone

cannot untangle certain manifolds, therefore requiring broader hidden layer(s).

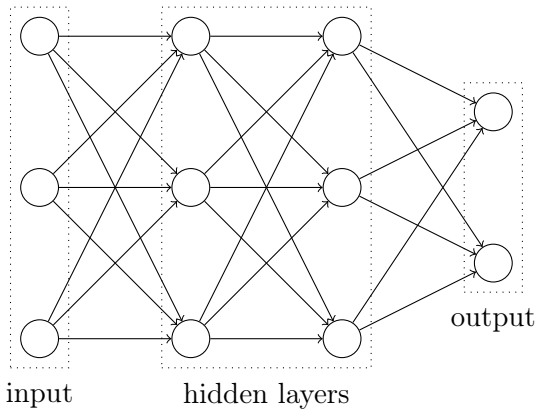


Figure 2.10: MLP with 3 inputs and 2 hidden layer with each 3 neurons and 2 output neurons.

The output of a single neuron in the output layer is:

$$\begin{aligned} \hat{Y}_i\left(x_i\right) &= \sigma_o\left(\sum_j w_{ij}^n h_{nj}\right) \\ h_{nj} &= \sigma_h\left(\sum_k w_{ij}^{n-1} h_{nj}\right) \ , \end{aligned} \tag{2.19}$$

which can also be written as in Equation 2.20. Note that the output of an MLP consists exactly of the composition of several perceptrons. Important to notice that the biases work exactly as for the perceptron, *i.e.* each layer can be extended with a "bias neuron" which has a unitary input, and each of the weights represent the bias for each subsequent layer neuron.

$$\hat{Y}_i\left(x_i\right) = \sigma_o\left(\sum_j w_{ij}^n \sigma_h\left(\sum_k w_{ij}^{n-1} h_{nj}\right)\right) \tag{2.20}$$

Back-propagation Algorithm for an MLP

The back-propagation algorithm is minimizing the error function by using a learning method and efficiently computing the chain rule of derivatives. The basic version was developed in 1986 by [17]. In the following algorithm, we describe this idea, by starting with a random initialization of all weights and biases.

1. Select an input pattern μ and define

$$X_k^0 = x_k\left(\mu\right) \ .$$

2. Forward propagate X_k^0, calculating and composing all outputs of each neuron as described at Equation 2.19
3. Calculate the final estimation error (δ) at layer M, and back propagate it to each neuron, at every layer (from M^{th} to 1^{st}):

$$\begin{aligned} \delta_i^M &= {\sigma'}_i^M \left(\sum_j w_{ij}^M h_j^{M-1} \right) \left(y_i(\mu) - \hat{y}_i^M \right) \\ \delta_i^{M-1} &= {\sigma'}_i^{M-1} \left(\sum_j w_{ij}^{M-1} h_j^{M-2} \right) \sum_j w_{ji}^M \delta_j^M \ . \end{aligned} \tag{2.21}$$

4. Update the weights from last to first layer (where $M = 1$):

$$w_{ij}^M(t+1) = w_{ij}^M(t) + \eta \delta_i^M h_j^{M-1} \ . \tag{2.22}$$

5. Repeat for the next input μ pattern, until the whole training set is performed.

"Back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using the gradient" according to [14]. Therefore the computational cost of the back-propagation is depended on the learning algorithm. The above example is the gradient descend learning procedure by updating the weights after each pattern (or input). Another approach is the stochastic gradient descend with the difference that the weight update is averaged over a user-defined number of input patterns. Such a combined pattern is called mini-batch because it consists of 10 to 50 input patterns, forming a batch.

2.2 Convolutional Neural Networks

2.2.1 Convolution

As an introduction to Convolutional Neural Networks (CNNs), it is crucial to have a deeper understanding of both mathematical meaning, definitions, and properties, but also to verify the results obtained by its applications, and consequences related to the field of image processing.

Convolutions can be intuitively understood as the total likelihood of two sequential events resulting in an expected third one. For instance, let us consider a golfing example. We have a hole with pair 3, of which we would like to make a birdie (*i.e.* using only two shots to put the ball in the hole). If shoot one is given by event $f(a)$, shoot two by $g(b)$, we can compute:

$$C = f(a) \cdot g(b) \ , \tag{2.23}$$

with C being the result birdie event. However, many other different combinations of two shoots also results in a birdie. Consequently the sum of all events resulting into a birdie can be obtained by:

$$C = \sum_{a+b=c} f(a)\, g(b) \ . \tag{2.24}$$

If we rewrite $b = c - a$, we end up at the definition of convolution given by Equation 2.25, representing in this example the sum of all possible shots resulting in a birdie.

$$(f * g)(c) = \sum_{a} f(a)\, g(c - a) \tag{2.25}$$

From the golfing example, we could have also inferred some properties of convolution as: associativity (also with real scalars), commutativity, and distributivity for instance. Properties of which, when applied to the CNN context will result in a fundamental property: dissociation between object spacial position, orientation, image brightness and such to the final classification results, when in comparisons to equivalent MLP only architectures.

Convolutions are applied in many different areas, from probability and statistics (as the golfing example), to optics, signal processing, and many other areas whenever superposition of two signals or impulse response needs to be calculated. In the context of CNNs, we could visualize the results of a two-dimensional convolution, defined by:

$$S(i, j) = K * I(i, j) = \sum_{m} \sum_{n} I(i - m, j - n) \cdot K(m, n) \tag{2.26}$$

and illustrated in Figure 2.11. The kernel moves across the input image to receive the output feature map. Such a convolution is performed on a real image in Figure 2.12.

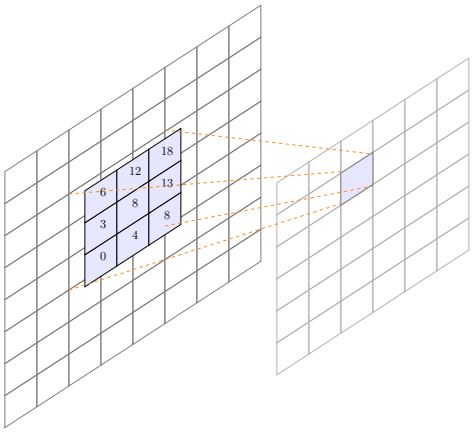
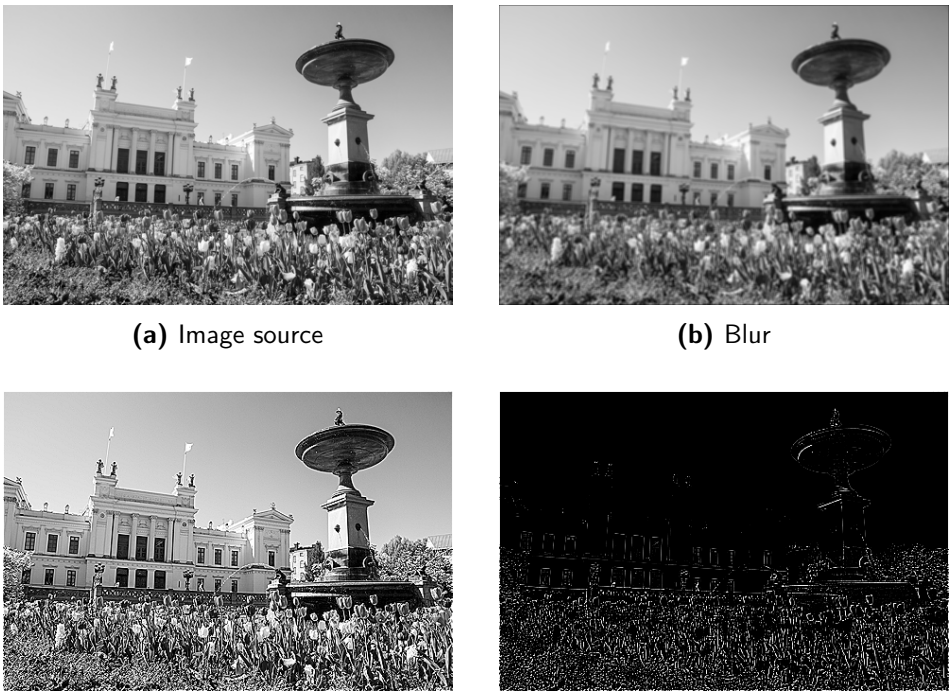


Figure 2.11: Input image 8x8, convolution with a 3x3 kernel and resulting 6x6 matrix.



(a) Image source **(b)** Blur

(c) Sharpen **(d)** Edge detection

Figure 2.12: Convolution filter examples on a gray-scale image of Lund University.

It is important to notice, that by performing this same operation, with slightly different kernels the convolution output as seen in Figure 2.12d can change dramatically, or merely reducing focus of the image, resulting in a smoothing effect as in Figure 2.12b or even sharpening the details as in Figure 2.12c.

The kernels utilized were:

$$\text{blur} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \tag{2.27}$$

$$\text{sharpen} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \tag{2.28}$$

$$\text{edge detection} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 7 & -1 \\ -1 & -1 & -1 \end{bmatrix}. \tag{2.29}$$

A noteworthy aspect of such diverse results applied by CNNs is the ability of self-learning and selection of the filters at each convolutional layer. The CNN after back-propagation will implement a composition of such filters (or many others),

resulting in feature maps that can successfully be classified by an MLP robustly to some image variations as described.

The two dimensional cases can be extended to a volumetric convolution for multi channel images illustrated by Figure 2.13.

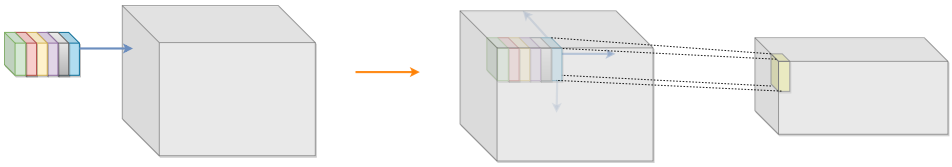


Figure 2.13: Example of 3-D convolution.

Notice that the 3-D kernel, in this case, can be seen as a stack of six 2-D kernels. Each valid element in the final result consists of the sum of each individual 2-D kernel convolved with the respective 2-D portion of the image.

2.2.2 Max-pool

The final component in order to fully build a convolutional neural network is the max-pool. The NxN max-pool operation is performed by taking the maximum element of a NxN sub-matrix either straight from the convolution or after the feature map (*i.e.* after convolution and activation). Max-pool is usually with stride N, meaning that once a max-pool is performed the next one will take place N+1 elements away, resulting in no overlap of the max-pool "window".

The main goal of the max-pool is to further ensure stability on the classification, once that small translations of the input will not affect the pooled result. Yet another convenient feature of pooling, in general, is the down-sampling effect. For instance the result of a MxM after a NxN max-pool is: M/NxM/N image. Figure 2.14 shows an example of a 2x2 max-pool of the previous showed blurred image in Figure 2.12b, where down-sampling can be observed.

Figure 2.14: Visualization of 2x2 max-pool.

To summarize, now that all the CNN building blocks are introduced, their output size can be calculated depending on hyperparameters (not directly trainable parameters) as in Table 2.1 .

Table 2.1: Most used hyperparameters in a CNN context.

	Hyperparameters	Output Size
Convolution	• Kernel size (K) • Stride (S) • Zero Padding (P)	$\text{Conv}_o = \frac{W-K+2P}{S} + 1$
Max-pool	• Pool size (P_s) • Stride	$\text{MaxP}_o = \frac{W-P_s}{S} + 1$
Activation	• Relu • sig • tanh	same as input

Where W is the width of the input images, which will be always equal to the height, for the majority of the commonly available datasets, since images are usually square-shaped.

2.2.3 CNN Architecture

Now that we have already seen all the basic building blocks of CNN, we can compose them as shown in Figure 2.15, forming a convolutional neural network.

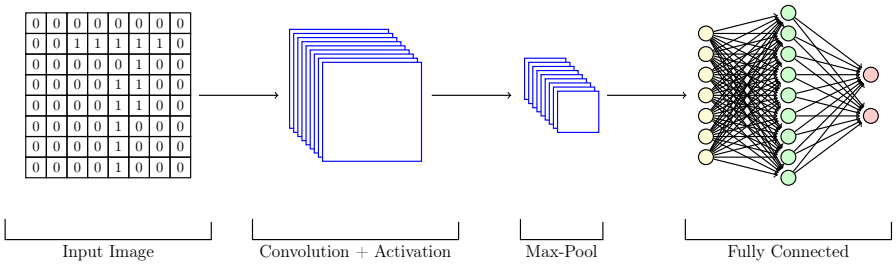


Figure 2.15: Example of a CNN architecture.

CNN can be seen as an arrangement of filters, which outputs are activated by a non-linear function, and down-sampled via max-pool (or equivalent), followed by an MLP. Alternatively it can be seen as *equivalent sparsely connected MLP with shared weights*. Sparsely connected in the sense that the receptive field of each convolution neuron equivalent MLP receives a much smaller input, given by the size of the kernel, instead of the full-sized image as it would have been for a fully connected MLP instead. Shared weights in the sense that all the "convolutional neuron" shares the same weights within the layer as in Figure 2.16 illustrated.

A convolution neural network can be formed by the combination of several convolution layers stacked, each being formed by convolution, activation, and

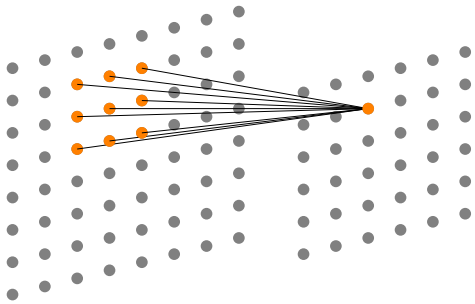


Figure 2.16: Convolution illustrated as MLP.

max-pool. As described by [9], the initial convolution layers learn the simpler edges of the images, arranging it into more complex and meaningful structures at the deeper hidden-layers of convolution layers, which are then classified by the MLP.

Chapter 3

Reference Model and Back-propagation

3.1 Implementation Strategy

This chapter will describe the dataset chosen as a benchmark for this thesis, together with a reference model selection, and an overview of the state-of-the-art in machine learning for such applications.

Since this project aims to accelerate only forward-propagation of convolutional neural networks, one needs to justify such an approach. A reasonable reason for not including also a hardware implementation for back-propagation is that for battery-powered applications, such as smart sensors, mobile phones, and wearables, training of such task-specific networks will be done offline, and the weights or architectural updates would be streamed by the network, in order to extend battery life. A large number of applications could not rely on forward-propagation being performed on cloud, as autonomous vehicles driving without available internet, for instance. In general, any mobile device offering AI features should be able to perform its actions locally, especially when controlling devices that can put at risk human-life. Newer parameters or architectures are less crucial and could be implemented as a software update, which can be performed at any time.

Another guideline followed by this project is to be able to accelerate any CNN architecture. As it will be seen, even for simpler datasets (as the one utilized for this thesis) the model's architectural characteristics can change dramatically if one aims to reach higher accuracies, or being flexible enough to solve broader range of problems. Therefore our vision, as for this project, is to accelerate the most basic operations involved with CNNs (or at least the typical case where most computational power/time is spent), and by only focusing on these, being flexible enough to be relevant and useful most of the time independently of any particular case or application.

Such methodology, of focusing on accelerating one of the building blocks aims to address the software nature of the Machine Learning (ML) problems. Given a scenario where there is no superhuman capability yet on solving the specific case, any model is susceptible of being changed entirely and thus making any hard-wired solution obsolete from a moment to the other. For instance, an MLP able to classify two objects, cannot tell anything about a third, without the further addition of neurons (and possible change of other hyperparameters). Additionally let us consider a hardware implementation of AlexNet used for autonomous vehicles,

having an impressive top-error rate of 37.5% on ImageNet 2012 [9] . A simple "update" on AlexNet capable of reducing this error-rate by for instance additional 10% most likely may require profound changes not only in the parameters values, but changing the configuration on the amount of layers, convolutions and max-pooling sizes, and other arrangements. Therefore while there is no superhuman solution for a given problem, fully optimized hardware-fixed solutions may not be able to cope with simple architectural changes, and thus at risk of being fastly outdated. For such fast-paced scenario a rather simpler alternative is the usage of GPUs (programmable, reliable, compatible with many APIs, and well known).

What seems to be relevant still in the future, demanding hardware solutions are the basic operations such as convolutions, multiplications, additions, and other fundamental building blocks. An underlying problem when considering those are still the classic bottlenecks as computational or memory limited systems.

3.2 MNIST Dataset

Before detailing further the implemented system, let us discuss more on the utilized reference model, and application.

The reference application addressed in this thesis is the Modified National Institute of Standards and Technology (MNIST) dataset. However, the implemented system is flexible enough to perform as well in any other application. MNIST was just chosen for convenience, as many people in the field benchmark their implementations using it. This dataset consists of labeled, size-normalized and centered images of digits 0 to 9. The image dimension is fixed to 28x28 pixels. The total number of examples is 60,000 training images and 10,000 for validation.[18]

A class visualization of the MNIST dataset can be seen in Figure 3.1. Such classification space representation can be obtained by the usage of the T-Distributed Stochastic Neighbor Embedding (T-SNE). Briefly T-SNE allow us to visualize high-dimensional data into lower dimensions while preserving similarity between different groups clustering, when in comparison to other techniques such as Principal Component Analysis (PCA).

As it can seen in Figure 3.1 our goal is to train a CNN which will find out a hyperplane which linearly separates the classes, or does its best to reach as close as possible to such linear separability.

3.2.1 State-of-the-art MNIST Classification Error

Convolutional neural networks as expected, serves as the most powerful technique as of today, in order to classify MNIST data with the highest accuracies. Note that CNNs are performing as good as, or even outperforming human beings in such dataset. According to [20], human's error rates are $\approx 0.2\%$, virtually the same as the state-of-the-art record as seen in Table 3.1.

Results obtained by [20] utilized an architecture given by: 35(1x29x29-20C4-MP2-40C5-MP3-150N-10N DNN). Meaning one input 29x29 is fed into the first convolutional layer, containing 20 4x4 kernels, followed by a tanh activation of the convolutions, non-overlapping 2x2 max pooling, 40 5x5 kernels in the second convolutional layer, tanh activation, 3x3 max-pool, all converging into a 150

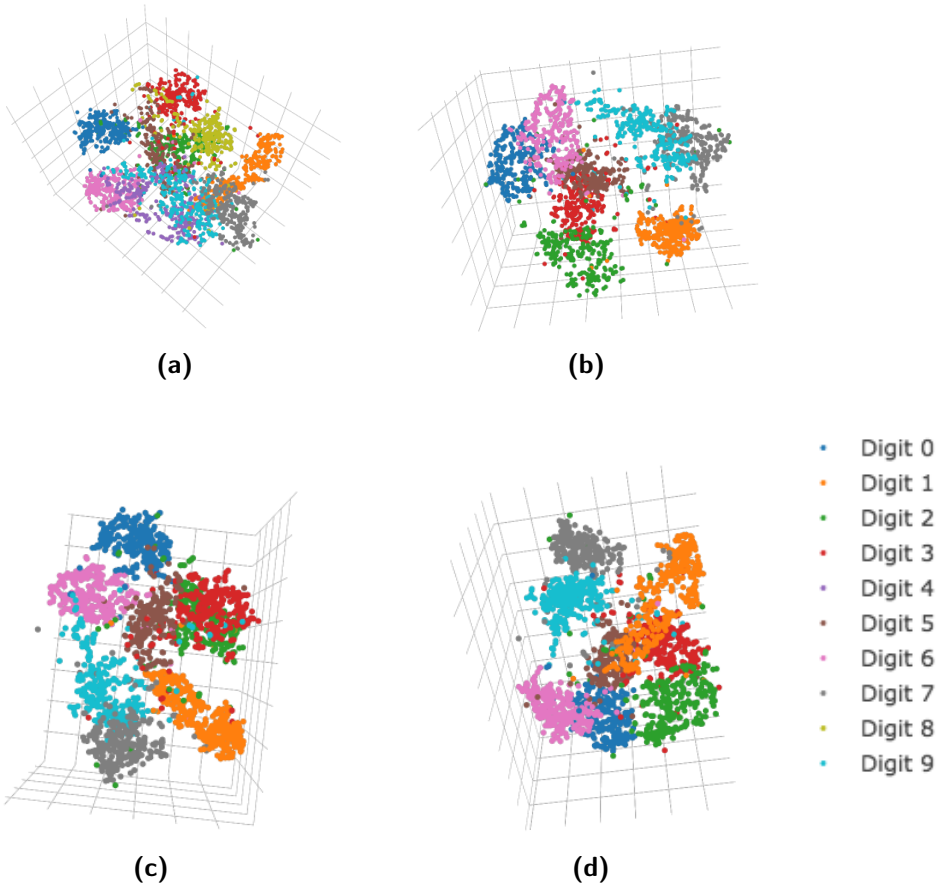


Figure 3.1: MNIST Classes decision hypercube, created with t-SNE Explorer [19].

neurons - tanh - 10 neuron - softmax MLP. The work [20] does this for 35 parallel CNNs described above, receiving each a slightly modified input obtained by different normalization. Finally, the classification result is obtained by averaging the individual outputs, in order to make a single classification.

We estimate for the architecture of [20] that roughly 3,000 weights are needed for each of those CNNs, resulting in total of roughly 105,000 parameters, most likely 32-bit floating points. This massive configuration improved that time state-of-the-art by impressively 34%, meaning that the anterior top-1 record error improved from 0.35% to the obtained 0.23%.

A simple model trained by us with as few as 222 parameters had 8% top-1 error rate. This fact illustrates that after a certain threshold the algorithm/architectural expenses explode in order to further improve a decimal of accuracy. This "little" improvements could be seen as irrelevant, however such specific applications may control devices handling lives, where reliability is crucial. Moreover, for this handwritten dataset, as mentioned before is roughly as good as human level

Table 3.1: Result of the top-1 classification error for the MNIST dataset state-of-the-art as from 2016. [21]

Result	Method
0.21 %	Regularization of Neural Networks using DropConnect [22]
0.23 %	Multi-column Deep Neural Networks for Image Classication [20]
0.23 %	APAC: Augmented Pattern Classification with Neural Networks [23]
0.24 %	Batch-normalized Maxout Network in Network [24]
0.29 %	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree [25]
0.31 %	Recurrent Convolutional Neural Network for Object Recognition
0.31 %	On the Importance of Normalisation Layers in Deep Learning With Piecewise Linear Activation Units [26]
0.32 %	Fractional Max-Pooling [27]
0.33 %	Competitive Multi-scale Convolution [28]
0.35 %	Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition [29]

classifications, but for more challenging datasets as ImageNet the top-1 error best results are at 16.5% [30] (obtained by Inception V4 architecture as of 2016). Impressive results, but still yet a long way to outperform human beings.

It is relevant to stress that improving the top-1 error rates at ImageNet from [30], may require total architectural changes of the inception V4 to the point that a hardware accelerated circuit working well for this, might not be able to execute all newer architectures, if a higher flexibility was not taken into consideration while optimizing only for the Inception architecture.

3.3 Tensorflow Model

In order to test our system, first task was to obtain a model, capable of performing reasonably well at the MNIST dataset. For this, a Tensorflow model was created, back-propagation performed and finally the weights were obtained.

Tensorflow is an open-source library, created by Google, to nurture development of ML applications. It has many built-in functions, where one can research on a mathematical, or algorithmic level implementations of neural networks for solving one's own datasets or the many other available options. Tensorflow is also very convenient as it offers easy integration of GPUs or multicore acceleration if available by the user.

Our implemented reference model trained with Tensorflow has the following architecture: 1x28x28 - 2C5 - MP2 - 10C5 - MP2 - 160 - 10. Activations are rectifier linear units, and final classification layer is implement by a softmax layer.

The model has a total of 1,427 parameters, trained with 32-bit floating point and resulting in 0.9729 final accuracy after 30 epochs.

The Python code in Listing 3.1 shows the implementation of such an architecture in Tensorflow.

Listing 3.1: Reference Model Architecture.

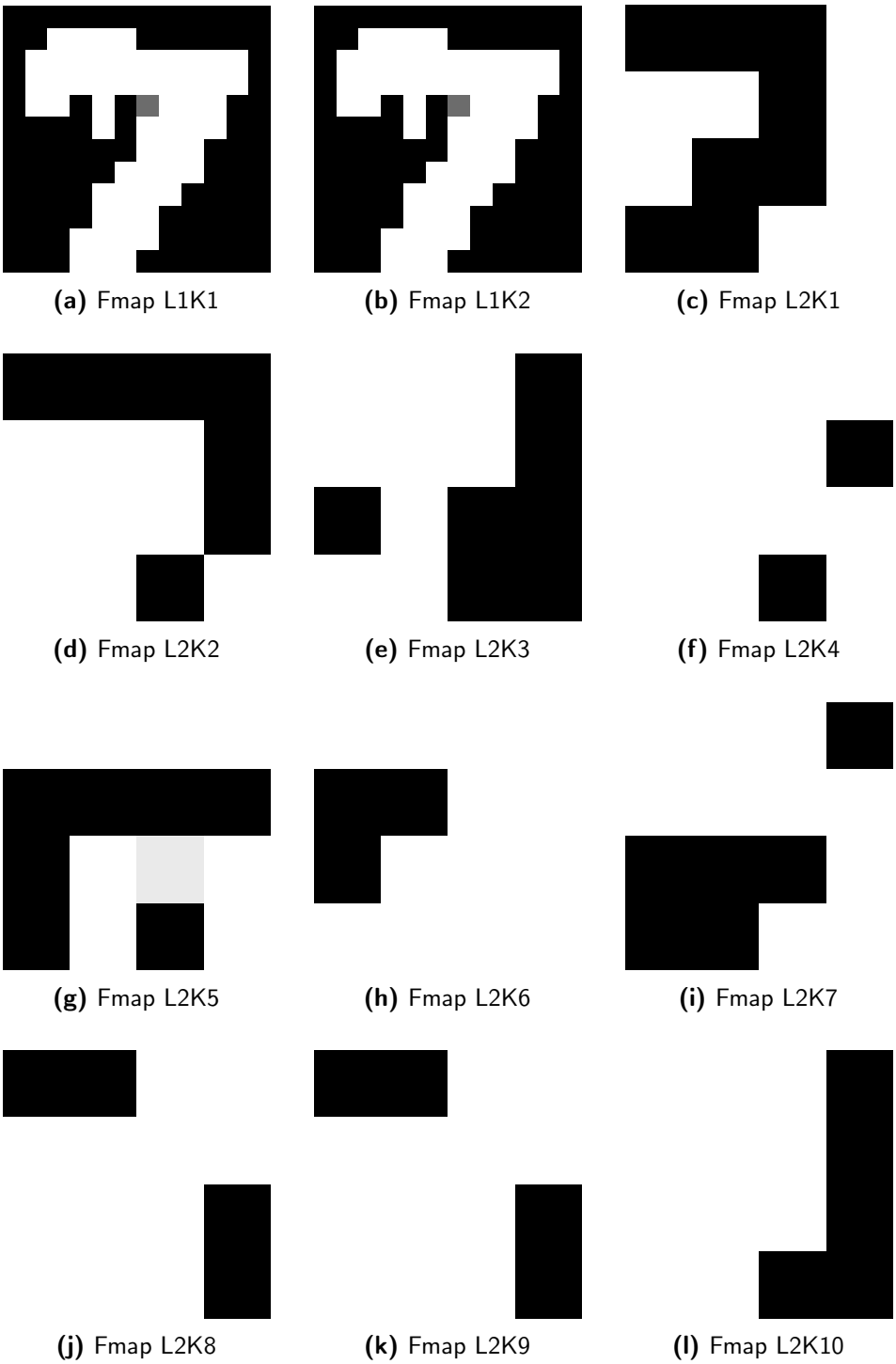
```
  model = Sequential()

# Layer 1 - Convolution 5x5 kernel
  model.add(Conv2D(2, kernel_size=(5,5), activation=None, input_shape=
      input_shape))
# Layer 1 - Maxpool 2x2
  model.add(MaxPooling2D(pool_size=(2,2)))
# Layer 1 - Activation ReLU
  model.add(Activation('relu'))

# Layer 2 - Convolution 5x5 kernel
  model.add(Conv2D(10, kernel_size=(5,5), activation=None, input_shape=
      input_shape))
# Layer 2 - Maxpool 2x2
  model.add(MaxPooling2D(pool_size=(2,2)))
# Layer 2 - Activation ReLU
  model.add(Activation('relu'))

# Layer 3 - Flatten and fully connected
  model.add(Flatten())
  model.add(Dense(5, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
```

3.3.1 Visualization of the Forward-Propagation

As a real example of the forward-propagation using our reference CNN model, the image in Figure 3.2 was fed to the CNN and Figure 3.3 illustrates the feature maps after each convolution. Notice that the image gets more blurred after each layer, and it seems that at the second convolution layer feature maps are focusing on the shapes of the different edges present in the original image. Serving as the input for the MLP, which will base its classification decision on the presented edges.

Figure 3.2: Input image from the MNIST dataset fed to the reference model.

After this project's reference model guidelines, architecture, and behaviour was discussed, we will introduce Tensilica processor's design.



(a) Fmap L1K1 **(b)** Fmap L1K2 **(c)** Fmap L2K1

(d) Fmap L2K2 **(e)** Fmap L2K3 **(f)** Fmap L2K4

(g) Fmap L2K5 **(h)** Fmap L2K6 **(i)** Fmap L2K7

(j) Fmap L2K8 **(k)** Fmap L2K9 **(l)** Fmap L2K10

Figure 3.3: Feature Maps visualization after convolution sorted by layer and kernel, given an MNIST image containing the digit seven.

Chapter 4
Processor Architectures

4.1 Xtensa Processor

Tensilica is a part of the Cadence Design Systems and provides a configurable and extendable processor for system-on-chip (SoC) applications. It gives the possibility to design a customizable processor for specific needs as high-performance, flexibility, and low power by providing an automatic creation of Register Transfer Level (RTL), compiler, and other back-end and verification scripts. [31]

In this thesis, we focused on the low power implementation by reducing the processor to our need and adding application-specific instructions. In the following sections, we will discuss the architecture of the Xtensa core and the changes we have considered. Also, we will briefly describe the development flow for a Tensilica based system.

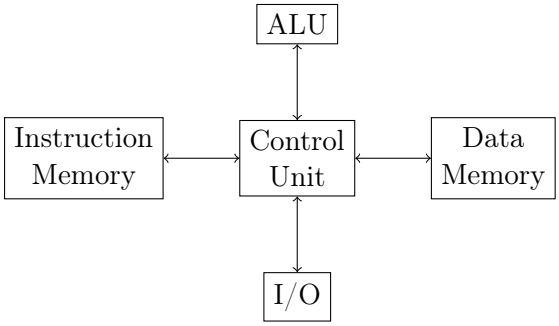


Figure 4.1: Processor Harvard architecture.

4.1.1 Xtensa Architecture

The processor architecture is based on the Harvard architecture where the instruction- and data-memory is physically separated like shown in Figure 4.1. This has the advantage of parallel memory access by fetching an instruction from and writing back the processed data to the memories. Depending on the memory speed a 5- or 7-stage pipeline can be selected. A bigger pipeline increases the number of instructions per time unit and could increase the program speed, but with

limitations such as latency, imbalance of the pipeline stages, and hazards. Such structural, data and control hazards lead to necessary stalls of the pipeline which could end up in a lower performance. [32]

The core has a 32-bit Arithmetic Logic Unit (ALU) with up to 64 general-purpose registers. The Instruction Set Architecture (ISA) is based on the Reduced Instruction Set Computer (RISC), nevertheless it may be extended with a Very Long Instruction Word (VLIW) for parallel instruction execution that are implemented via Tensilica Instruction Extension (TIE).

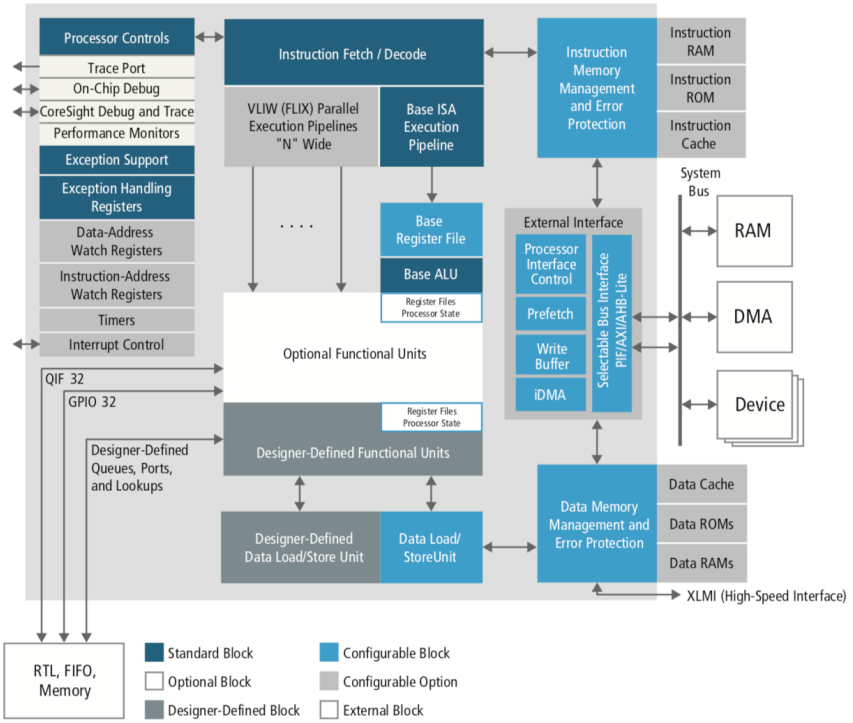


Figure 4.2: Xtensa architecture and possibilities. [31]

Figure 4.2 shows the provided standard, configurable and optional blocks. As shown there are two tightly coupled memories for instructions and data. In our case, we have implemented one instruction memory (IRAM) and two data memories (DRAM) with a word length of 32 bits to match with the 32-bit ALU. These memories are accessed by load/store units. Here the user can configure the core utilizing one or two units. Two units are necessary if a significant amount of data needs to be moved. Besides, the memories can be externally accessed through the external interface, called Processor Interface (PIF) with the option of implementing an Advanced Extensible Interface (AXI) bus. This interface connects the processor with the system memory and other peripherals.

One of the main advantages of Tensilica is the possibility of extending the processor through TIE for example to add instructions, or customized input/output

interfaces. Those extensions are described in chapter 4.1.2. For verification of all the features after fabrication on-chip debug and trace components can be optionally selected. This additional hardware enhances debuggability possibilities during verification of a fabricated device.

4.1.2 TIE

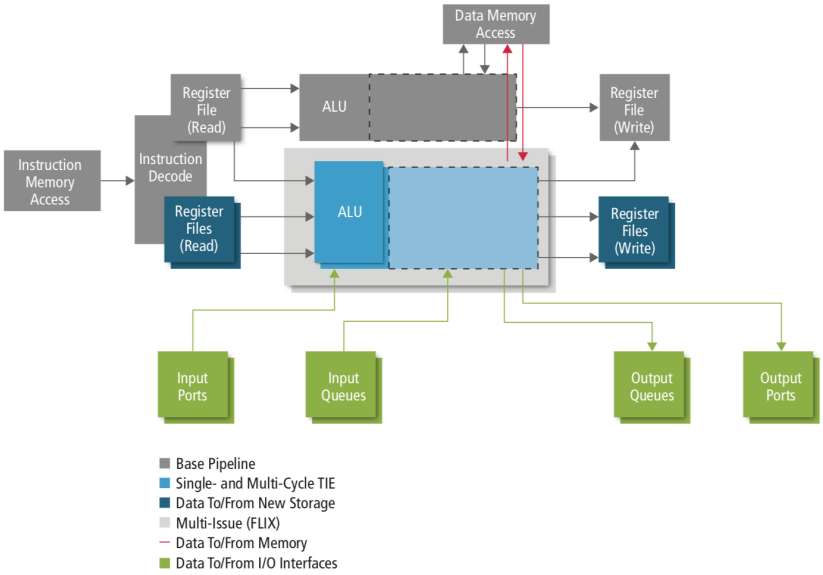


Figure 4.3: Xtensa instruction pipeline with extensions. [33]

TIE is a Tensilica specific language to extend the processor instruction and is based on the Verilog syntax. Figure 4.3 shows the effect of TIE. The basic ALU will not be affected, and another ALU will be implemented in parallel. In the following chapter, we will explain the possible optimization techniques in TIE.

Fusion combines multiple operations to a single instruction to increase the performance of a loop, for instance. Another method is called Flexible Length Instruction Extensions (FLIX). It uses a variable instruction length VLIW to issue multiple operations in one cycle by packing it in up to 128-bit instructions. This performance improvement of issuing in parallel multiple operations has the drawback of additional hardware cost. The compiler selects the FLIX instruction so that it increases the performance of the code, while avoiding usage of NOP (no operation) instructions.

Another powerful performance enhancement technique is to vectorize the processor by using Single Instruction, Multiple Data (SIMD) to increase the data-level parallelism. Thus up to 1024 bits and 128 entries of own specific register files can be added. These need to be extended by custom instructions to perform operations, for example, addition and multiplication, to handle the new registers file. In the following example at Listing 4.1 we are creating a register file with 16 entries, and

each of them is 64 bit wide. Additionally, we implement a vectorized addition. Therefore the input signals A and B are split into 16-bit values and added together. The single results are stored in temporary signals, and these signals are using the *wire* command to indicate an intermediate type. The keyword *assign* is used to write to a real signal. Thus the SIMD result is the vector of all the temporary results of the calculations. Therefore it is vital to keep the right endianness as the processor and the other instructions. For the given example the performance increases by four times only because of the data-level parallelism.[33]

Listing 4.1: TIE example on parallel computation.

```
regfile rf64 64 16 name    // 64 bit wide with 16 entries

operation vec4add16 {out rf64 res, in rf64 A, in rf64 B} {}
{
    // split into the different bits and add
    wire [15:0] tmp0 = A[15: 0] + B[15: 0];
    wire [15:0] tmp1 = A[31:16] + B[31:16];
    wire [15:0] tmp2 = A[47:32] + B[47:32];
    wire [15:0] tmp3 = A[63:48] + B[63:48];

    // assign the temporary results to the result
    //(ATTENTION: endianness)
    assign res = { tmp3, tmp2, tmp1, tmp0};
}
```

As shown in Figure 4.3 there is also direct input/output (I/O) interface to the ALU via queues, ports and memory look-up. These I/O interfaces can be used to transfer data or control signals between the ALU and other RTL components or memories. Such direct connections are fast and do not need any address. The access is implemented via TIE to enhance the performance. A single instruction can access multiple I/O interfaces. Queues can be used to connect external synchronous FIFO (first-in-first-out) devices. During creation automatic handshake signals are added to access data. The look-up interface could be connected to an external memory such as ROM or RAM. Which could perform for instance an activation function look-up, instead of executing high complex calculations. The last ports are General-Purpose Input/Outputs (GPIOs) with up to 1024 bits wide connection to external RTL blocks. The output is written to specific registers that can be accessed from "outside" through *export state*. The input is connected via *import wire* that are visible from the processor. These ports are often used as control or status ports. Hence it is essential to know that the compiler does not reorder the TIE instructions differently and optimizing away the read and write instruction of the GPIO ports. The compiler needs additional information to guarantee the sequential order of critical actions. [33]

The implemented convolution accelerator close to the Xtensa core is controlled via GPIO ports and a specific instruction sends all the information per convolution in a single cycle. This reduces the setup time by four cycles per convolution.

4.1.3 Xtensa Methodology

The Xtensa hardware/software development is performed with the Eclipse based graphical user interface called Xtensa Xplorer IDE. From there C/C++ programs can be created and verified on different user-designed processors. Such processor can be created by using check-boxes and adding own specific instructions, as described before. Profiling the implemented algorithm to identify hotspots and compare different implementation is a very powerful feature of the Tensilica profiler. It is also an excellent way to compare the compiled assembly and C-code with information about the cycles per instruction. It also provides an idea about the performance degrading functions. Furthermore, the number of stalls and taken branches during the execution can be analyzed.

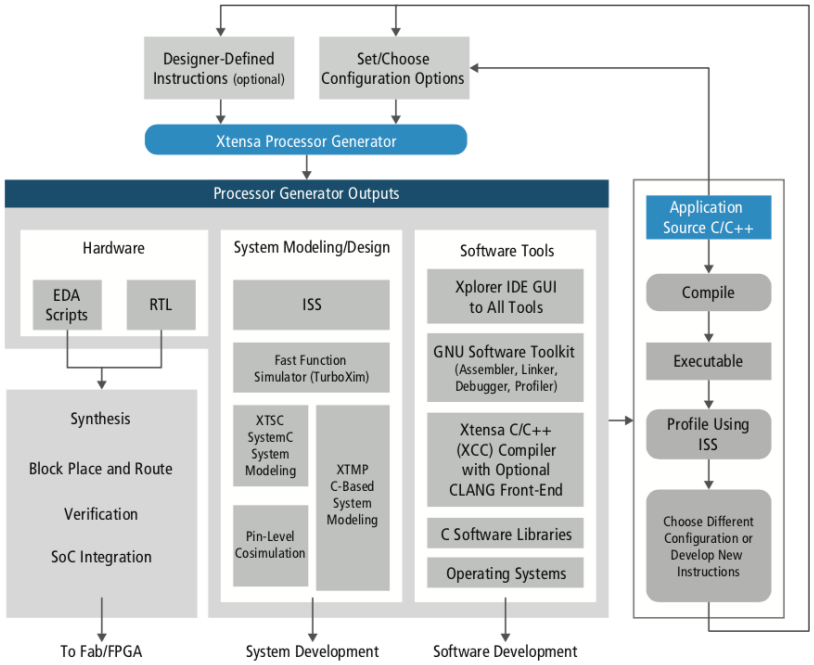


Figure 4.4: Development methodology of the Xtensa workflow.[31]

Figure 4.4 shows the development methodology of an Xtensa processor. Therefore, the user designs a C/C++ application and profiles to choose the processor and necessary optimizations for a specific application. The Xtensa Processor Generator (XPG) creates the specific RTL, EDA scripts for verification and synthesis based on the selected configuration. Also, the software development tools are generated to have an exclusive compiler for each processor, together with a system model and design development flow. This gives access to faster simulations than the RTL-level simulation for verification. To speed up the simulation time different scripts, for individual simulations of each interface, are provided. This hardware/software co-development gives the advantage to create a specific embedded system.[31]

4.2 RISC V Processor

An open-source alternative to Tensilica would be the RISC V processor, for example, the PULPino with the RI5CY core (single core). This version has an extended instruction set and is based on the RV32IMFC. Therefore we will describe the RISC V and the implementation of the hardware accelerator shortly.

This processor is a 4-stage pipelined 32-bit processor like shown in Figure 4.5 with instruction fetch, instruction decode, execution and write back stage. In the execution stage, there is a standard ALU supporting basic integer operations, extended with a multiplication and division unit for fixed point. Optionally, a floating point unit (FPU) is also available. For this project, we do not see any advantages by usage of floating point unit. Fixed point operations are more energy efficient and good enough to achieve high accuracy in ANNs.

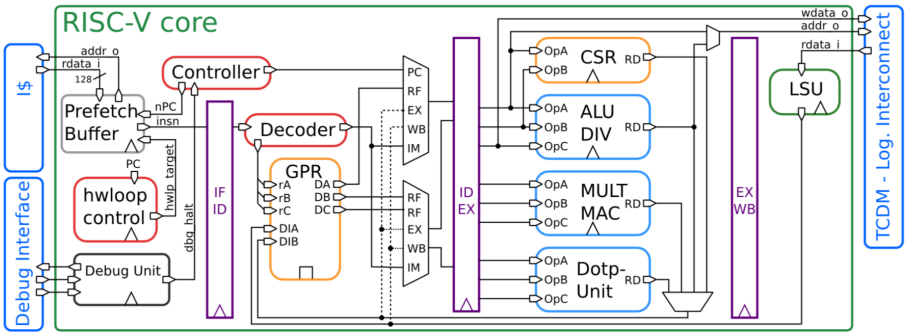


Figure 4.5: RISC V 4-stage pipeline. [34]

It also consists of two separated single port memories for instruction and data as shown in 4.6. The boot loader is stored in the boot ROM and can read a program through the Serial Peripheral Interface (SPI) to the instruction memory. The main bus is an AXI interconnection with a bridge to Advanced Peripheral Bus (APB). The advanced debug unit is connected via AXI with the processor to read out the internal registers and can also access the two RAMs and the APB. Also, there are components like GPIO, UART, timer and further features available. [34]

To connect a hardware accelerator to the RISC V processor the APB interconnet can be selected by changing the memory map. This connection can be used to control the hardware accelerator. To guarantee stable throughput direct memory access should be implemented. Besides, the convolution setup must to be done via the 32-bit APB interconnect, hence it is not possible to do it by a single instruction as in Tensilica. To indicate the end of such convolution the processor receives a ready signal via an interrupt, awakening the processor.

Using the RISC V has a trade-off, the processor is well known and open-source but adding own instructions and modifying the compiler may take considerable longer compared to the Tensilica processor. However, this processor may serve as a control unit of different hardware accelerators, which are performing the major part of the calculations. Such a system needs to consists of the basic operations

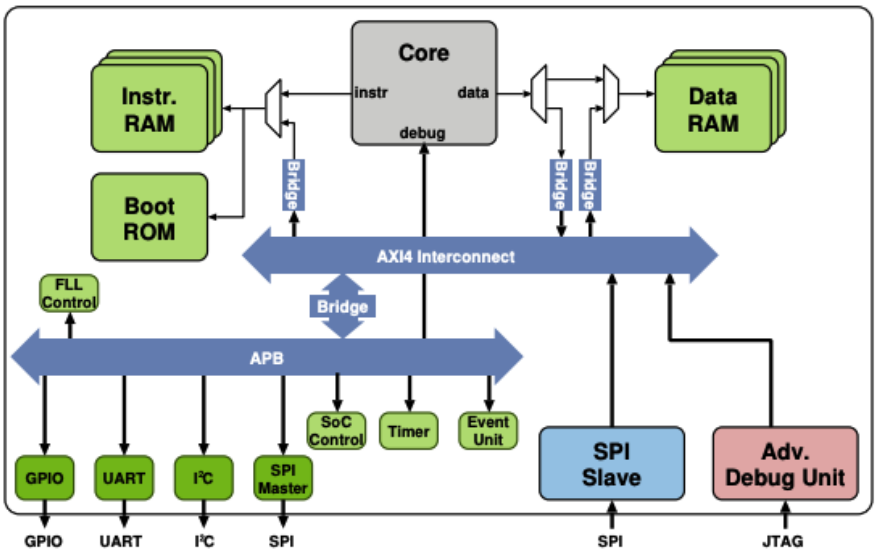


Figure 4.6: Overview of the PULPino architecture. [34]

of a CNN while keeping the flexibility to adjust the hyperparameters and the network architecture. For example, we are performing a convolution followed by an activation but without max-pooling. These different architecture possibilities have to be taken into account to design a sustainable ASIC.

Chapter 5
Convolution Hardware Accelerator

In the following chapter we describe the main computational bottleneck identified, and a proposed hardware architecture addressing it. Additionally, we decompose the convolution into a Finite Impulse Response (FIR) filter and derive the fast FIR implementations. Furthermore, we describe more details about the accelerator and finish by analyzing the implementation.

5.1 Identification of Computational Bottlenecks

After the selection of the reference model as described before, a C++ algorithm including all necessary functions for the CNN was developed to be executed by the Xtensa processor. The computation time on each of the functions are shown in Table 5.1.

Table 5.1: Time spent on each of the main building blocks of the reference model.

Function	**Time Spent (%)**	**Number of Calls**
Conv2D	28.17	22
Conv3D	0.32	10
Maxpool	0.32	22
Vsum	0.26	4
LoadModel	0.11	1
Relu	0.06	3
Fully Connected	0.02	1
Flatten	0.01	1
Softmax	0.00	1
Exp	0.00	20

For the reference model (1x28x28 - 2C5 - MP2 - 10C5 - MP2 - 160 - 10), 28.17% of the time is spent only in calculating the basic convolution alone. Not considering the memory allocation to store the results, else 66.86% of all computational time is spent in the convolution to perform this CNN model. The remaining time was spent performing many other processor's instructions, such as allocating/freeing

memory many times, and other auxiliary functions not directly related to the CNN functions.

Another important factor, that should be considered in the identification of the computational hotspot is the scalability issue as seen in Table 5.2. Note that by making a single image 4 times bigger, the convolution time (5x5) will increase by 46.85 times, and 188.69 if the image is 8x bigger. Moreover, by having a bigger image, the whole CNN computational time will be impacted by a much greater factor, since each individual convolution time will increase.

Table 5.2: Image dimension in pixels *vs* software only convolution computational time.

Image Size (pixels)	**28x28**	**112x112**	**224x224**
Image Increase (x)	1	4	8
Computational Time Increase (x)	1	46.85	188.69

A possible indication for such scaling complexity are the costly memory operations. For the reference model seven within the TOP-21 most issued instructions are memory instructions, clustered as shown in Table 5.3.

Table 5.3: Occurrence of the top-21 load/store instructions as a total of all issued instructions.

Instruction	Occurrences (%)
load 32-bit	30.0
store 32-bit	13.89
load 8-bit	1.79
Total	45.68

Therefore, it is expected that by usage of multiple load/store units one can significantly speed-up the final system because multiple of those instructions could be issued at the same clock cycle. A compromise is made by this possible strategy, once that from convolution layer to the next, the number of elements to be calculated is divided by four (by using 2x2 pooling). Consequently one has to find a number of load/store units to minimize the number of memory access cycles while minimizing the amount of no operations (when aiming for high efficiency).

Finally, for all the reasons previously described, it was clear that the main bottleneck for forward-propagating CNN are at a first moment the convolution operations, in terms of computational time and memory requirement. Convolutions in CNN will not only be the main bottleneck in general, if the input size is not big enough, or if the amount of convolutions performed are insignificant in comparison to the MLP (not the usual case for artificial vision applications).

5.2 Convolution as Fast FIR Filter

The convolution equation (2.26) can be written as:

$$S(i,j) = \sum_{m}^{M} \left(\sum_{n}^{N} I(i-m, j-n) \cdot K(m,n) \right) .$$

This can be transformed to a sum of M FIR filters with N-taps. One single FIR filter is given by

$$y(x) = h(n) \cdot x(n) = \sum_{i=0}^{N-1} i = h(i) \cdot x(n-i) , \tag{5.1}$$

with $n = 0, 1, 2, ..., \infty$. The FIR filter can be implemented as fast FIR filter by a strength reduction technique based on [35]. First of all we need a formulation of the parallel FIR filter using the polyphase decomposition. It starts by converting the equation into the discrete domain via z-transformation.

$$Y(z) = H(z) \cdot X(z) \tag{5.2}$$

The infinite input $X(z)$ and output $Y(z)$ sequence can be decomposed into even and odd numbers. Also for the filter coefficients H(z).

$$\begin{aligned} X(z) &= x(0) + x(1)z^{-1} + x(2)z^{-2} + ... \\ &= X_0(z^2) + z^{-1}X_1(z^2) \end{aligned} \tag{5.3}$$

$$\begin{aligned} Y(z) &= y(0) + y(1)z^{-1} + y(2)z^{-2} + ... \\ &= Y_0(z^2) + z^{-1}Y_1(z^2) \end{aligned} \tag{5.4}$$

$$H(z) = H_0(z^2) + z^{-1}H_1(z^2) \tag{5.5}$$

Consequently, the outputs can be computed as follows:

$$\begin{aligned} Y(z) = X_0(z^2)H_0(z^2) &+ z^{-1}(X_0(z^2)H_1(z^2) \\ &+ X_1(z^2)H_0(z^2)) + z^{-2}X_1(z^2)H1(z^2) \end{aligned} \tag{5.6}$$

$$\begin{aligned} Y_0(z^2) &= X_0(z^2)H_0(z^2) + z^{-2}X_1(z^2)H_1(z^2) \\ Y_1(z^2) &= X_0(z^2)H_1(z^2) + X_1(z^2)H_0(z^2) . \end{aligned} \tag{5.7}$$

The two output equations $Y_0(z^2)$ and $Y_1(z^2)$ are equivalent in the time domain to $y(2k)$ and $y(2k+1)$. This is a 2-parallel FIR filter with a throughput of two. However, this filter can be further optimized by reshaping the equation as followed.

$$Y_1 = X_0H_1 + X_1H_0 = (H_0 + H_1)(X_0 + X_1) - H_0X_0 - H_1X_1 . \tag{5.8}$$

Resulting in the following equations:

$$\begin{aligned} Y_0 &= H_0 X_0 + z^{-2} H_1 X_1 \\ Y_1 &= (H_0 + H_1)(X_0 + X_1) - H_0 X_0 - H_1 X_1 \ , \end{aligned} \tag{5.9}$$

implementing a fast FIR filter with two shared terms between the outputs. This reduces the number of multiplications from $2N$ to $2N - \frac{N}{L}$ where N is the number of taps and L the level of parallelism. The final 2-parallel filter can be written in the matrix from $Y_2 = Q_2 H_2 P_2 X_2$.

$$\begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & z^{-2} \\ -1 & 1 & -1 \end{pmatrix} \operatorname{diag} \begin{pmatrix} H_0 \\ H_0 + H_1 \\ H_1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \end{pmatrix} \tag{5.10}$$

In the previous part, we have derived the formula for the 2-parallel fast FIR filter by using the polyphase decomposition. The resulting block diagram of the implementation is shown in Figure 5.1. Here it is essential to understand that each H can contain subfilters so that any filter can be designed.

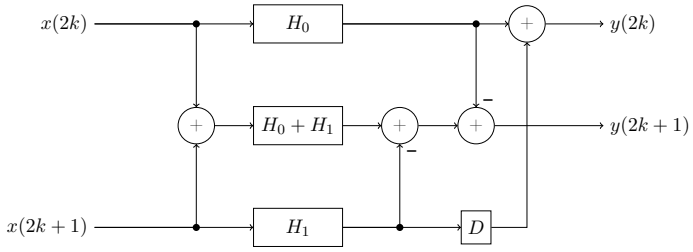


Figure 5.1: Implementation of 2-parallel Fast FIR filter.

5.3 Hardware Implementation and Integration

The previously derived fast FIR filter can be used to implement an efficient convolution hardware accelerator. We have focused on a kernel size of 5x5 because it is very often used in CNNs. It can also perform the 3x3 kernel by zero padding (adding zeros) the kernel.

Assume we have the following kernel:

$$k = \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} & k_{04} \\ k_{10} & k_{11} & k_{12} & k_{13} & k_{14} \\ k_{20} & k_{21} & k_{22} & k_{23} & k_{24} \\ k_{30} & k_{31} & k_{32} & k_{33} & k_{34} \\ k_{40} & k_{41} & k_{42} & k_{43} & k_{44} \end{bmatrix} . \tag{5.11}$$

Every FIR receives from each row of the kernel five coefficients. Consequently the coefficient for one filter is obtained by:

$$H_0 = [k_{00}\ k_{02}\ k_{04}]\ , \tag{5.12}$$
$$H_1 = [k_{01}\ k_{03}\ 0]\ , \tag{5.13}$$
$$H_0 + H_1 = [k_{00} + k_{01}\ k_{02} + k_{03}\ k_{04}]\ , \tag{5.14}$$

resulting into the following block diagram.

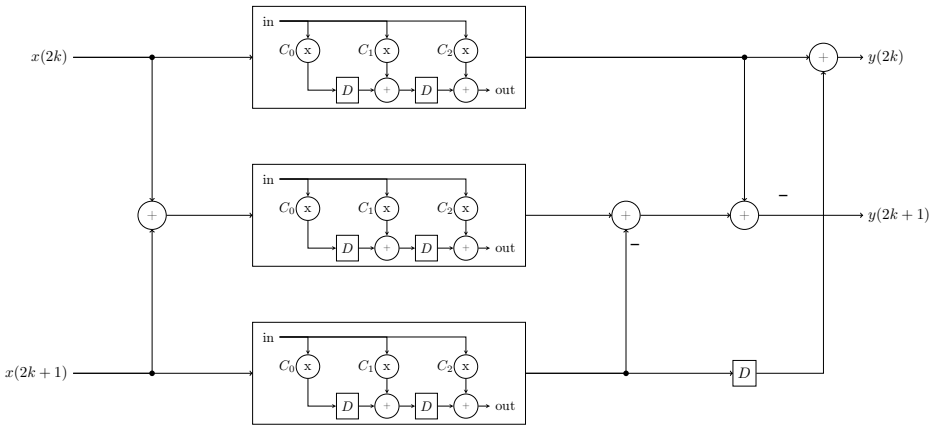


Figure 5.2: Fast FIR filter as part of a full convolution.

To perform the 5x5 kernel convolution five fast FIR units are needed, each unit is called here Fast FIR Computation Unit (FCU). Equation 5.1 shows the results of each FCU, these need to be summed together as shown in Figure 5.3. The input must be delayed between the FCU units to slide the kernel row-wise through the image. Therefore we implemented a controllable shift register.

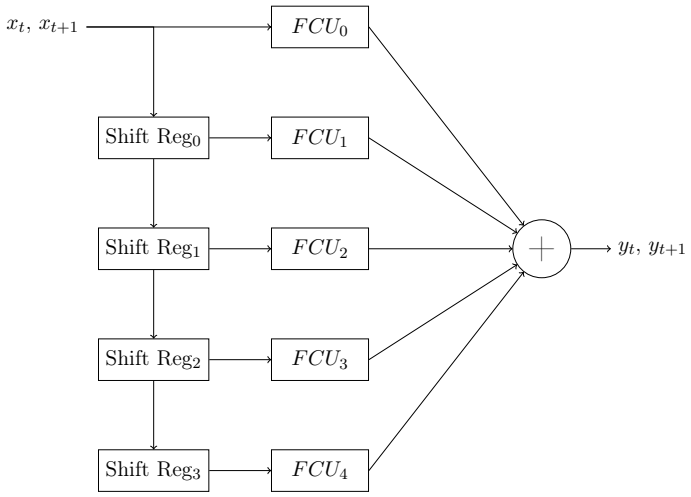


Figure 5.3: 5x5 convolution implemented with fast FIR filters.

This convolution processing unit is one of the main components of our hardware accelerator. In Figure 5.4 all the single components and connections are shown. The system reads from the memories through the memory interface, that is implemented as a crossbar. Therefore we need to have direct access to the memories by using, for example, a dual port memory. As designed the input image is expected in memory 0 and output in memory 1 or inversely. This results in the advantage of reduced usage of memories inside the accelerator. The kernel data is stored in a register file (kernel register) and configured during setup. The controller handles all the I/O signals of the system while controlling the data flow of the internal components.

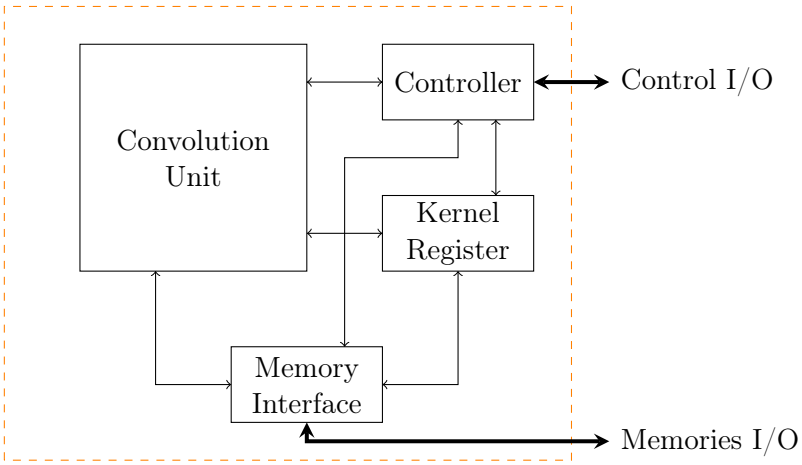


Figure 5.4: Block diagram of the hardware accelerator.

To understand the working routine of the accelerator it is fundamental to know the incoming signals. Depended on the connection to the processor the transfer of the setup information can be performed in a single or multi-cycle transfer. The input signals are

- Address input image
- Address kernel
- Address result
- Input image size
- Start-bit

The input information are stored in registers. When the start-bit occurs, the kernels and bias are configured by reading each element from memory (two 16-bit values per cycle), and some pre-calculations are performed so that during execution only simple counters are running. Afterwards, the input image values are read from memory. For each cycle two 16-bit elements are read, and after a latency, the results are written back to the second memory in pairs (total 32-bit). The convolution unit only gives the 'valid' convolution out. Meaning that an input image with the dimensions of 28x28 and kernel 5x5 will result in a 24x24 output

dimension. For some other algorithms zero padding is used to keep the same size for input and output. In our case the convolution is already used to down-sample the input so we are not using zero padding. After the convolution is done the ready signal is set to high and the processor reads or waits for an hardware interrupt. The approach selection is a design choice. Also the intermediate register are reset and the system is ready to perform the next convolution.

The hardware accelerator is implemented with 16 bit fixed point in a Q8.8 format. This binary representation was selected because of its small RMS of circa 0.0075 compared to smaller bit widths. This error appears due to truncation in the FCU and is depended on the input and kernel data. It is crucial to notice that this error could be further reduced by a fixed point neural network training algorithm. Also, such training could lower the requirement of a Q8.8 format. The performance of this implementation is one of the main advantages compared to other solutions. It needs 18 cycles to set up the kernel, the bias, and the counters. The cycle count for the convolution can be calculated by:

$$cycles = \frac{M \cdot N}{2} \ , \tag{5.15}$$

where M and N are the dimensions of the input image.

A short example to MNIST: input image dimensions of $28x28$, therefore it will need 392 cycles for the convolution and 18 for the setup. Thus, the full hardware accelerator needs 410 cycles to compute the convolution of an MNIST image. This cycle count scales up linearly. An image of $256x256$ dimension will result in 32786 cycles.

The hardware accelerator was synthesized for 200 MHz and 1 GHz without any memory. The worst negative slack went down to $0.0ps$. This shows that the convolution hardware accelerator can also perform high-frequency computations, with the cost of a higher power consumption. The leakage was 40% higher compared to the 200 MHz implementation. The frequency could be selected so that the system can run in real time for a specific application.

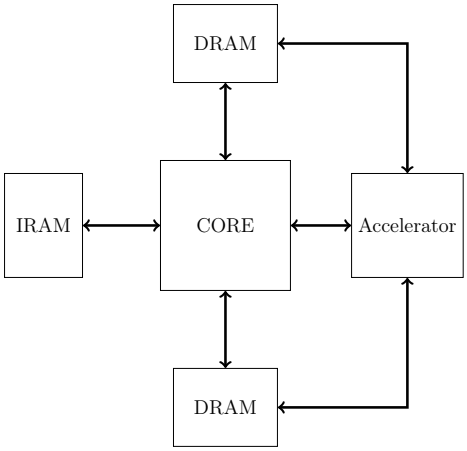


Figure 5.5: Integration of the hardware accelerator with the core.

Figure 5.5 illustrates the integration of the accelerator with the core and the tightly coupled SRAMs from Xenergic. The data memories have one dedicated port for the core and another one for the accelerator. As described before, the accelerator is controlled via the GPIO interface with an own instruction in the Xtensa core.

5.4 Hardware/Software Implementation Results

After the hardware accelerator was implemented and verified, together with the selected Xtensa core, an estimation of the new computation time per function was made as show in Table 5.4. The following results are obtained by estimation of the synthesized accelerator and processor estimations. The final system was block-synthesized, but co-simulations could not be performed, due to tool complications.

Table 5.4: Estimated new computation time of the hardware/software co-design. Final speed-up is estimated in 3 times faster than the software only approach.

Function	**Time Spent (%)**	**Number of Calls**
Conv3D	0.98	10
Maxpool	0.98	22
Vsum	0.79	4
LoadModel	0.36	1
Relu	0.19	3
Fully Connected	0.07	1
Conv2D	0.05	22
Flatten	0.05	1
Softmax	0.01	1
Exp	0.00	20

Notice that the main convolution function, initially implemented in software, was reduced from 28.17% to 0.05%, aided by the hardware accelerator. The other building blocks have their significance slightly changed, once that the new total number of clock cycles are reduced by three times.

No other performance limiting function that would call for a further optimization was observed. We rather detect in the reference model a high number of processes taking roughly one percent or less of the total computational time, summing up to a third of what was the software-only number of cycles. This lately observed processor overhead must still be further investigated in future work.

Finally, the expected energy/classification is:

$$\frac{E}{Class} = 0.092 \quad mJ/class \quad (@200MHz) \ ,$$

yielding at 87 classifications per second, with an estimated accuracy of 97%.

Table 5.5: State-of-the-art benchmarks compared to the present work.

	Bankman [36]		**IBM TrueNorth [37]**	**VLSI '17[38]**	**VLSI '17[39]**	**This work**
Technology	28 nm		28 nm	65 nm	40 nm	28nm
Algorithm	CNN		CNN	DNN	LCA	CNN
Dataset	CIFAR-10	CIFAR-10	CIFAR-10	MNIST	MNIST	MNIST
Precision [bits] (Weights, Activation)	(1,1)		(1.6,1)	(1.6,1)	(4,1)	(16,16)
Supply [V]	$V_{NEU} = 0.6$ $V_{COMP} = 0.8$ $V_{DD} = 0.8$ $V_{MEM} = 0.8$	$V_{NEU} = 0.6$ $V_{COMP} = 0.8$ $V_{DD} = 0.6$ $V_{MEM} = 0.53$	1.0	0.55 - 1.0	0.9	0.9
Classification Accuracy [%]	86.05	85.69	83.41	90.1	88	~ 97
Energy per Classification [μJ]	3.79	2.61	164	0.28 - 0.73	0.050	92
Power [mW]	0.899	0.094	204.4	50 - 600	87	7.88
Frame Rate [FPS]	237	36	1249	800 K - 3280 K	1.7 M	87.5

Table 5.5 shows prior relevant results in the field. First of all the dataset MNIST is gray-scale and CIFAR-10 (dataset containing ten different classes) contains colored images (RGB-format) with size 32x32 pixels. CIFAR-10 state-of-the-art classification accuracy is at 96.5% [27]. A comparison of the present work with CIFAR-10 implementations is valid, since the incremental cost in the reference architecture can be seen as irrelevant (two more convolutions in total are needed, and weight updates).

The design in [36], using the CIFAR-10 dataset, is a mixed-signal binary CNN processor using XNORs instead of multiplications, due to weights and activations being represented by a single bit. Additionally the filter size of the convolution is fixed to 2x2 and has up to 256 channels. Although relevant to the field [36] in comparison to the present work, has no flexibility. As discussed before, such methodology will become a good alternative when an architecture able to solve one of the great problems in machine learning is discovered, until this moment flexibility is crucial.

A deeper look into [39] shows that they have designed an analog computation neuron and a digital communication feature. By using the Locally Competitive Algorithm (LCA) and reducing the weights to four bits, they could archive a energy consumption of $50nJ/classification$. Their impressive frame rate is due to the fact that the LCA algorithm is much less computational demanding than CNNs, together with the chip area and amount of "analog neurons" implemented. As discussed before, the TOP-10 most accurate algorithms for the MNIST dataset, as of today, implement CNNs. LCA algorithms are relevant, but have not yet been widely used for artificial vision applications.

[38] is an implementation parallel in-memory computation for binary/ternary deep neural network (DNN), basically a deep MLP. Reconfigurability is limited to types as full, dense, and sparse. Additionally the number of neurons per layer and the number of layers can be selected. So no deeper MLP architecture changes can be done. This chip has a power consumption of $0.6W$ at 400 MHz. Work

done by [38] is very promising in terms of DNNs, however the motivation for the creation of CNNs was to reduce the amount of parameters necessary for a DNN equivalent architecture. Additionally as discussed, CNNs are very robust to spatial changes and image noise, when in comparison to DNNs. Thus, although with a certain degree of flexibility, [38] methodology is not be as scalable and flexible as the present work.

IBM [37] developed and published chip TrueNorth is based on a network of neurosynaptic cores. Block-wise programmable connectivity builds up the full CNN architecture. Activation and weights are represented by 1.6 (average) and 1 bit. They could achieve an accuracy of 83.41% with a power of $204.4mW$ for a single TrueNorth chip. The accuracy could be increased by combining 8 TrueNorth chips to 89.32%. The interesting results about this work is that they used different benchmarks to analyze the system, and obtained an impressive frame-rate of 1249 FPS. The work by [37] is the one which is most similar to ours. It offers a higher degree of flexibility by the aforementioned block-wise connectivity and a high frame rate, at the cost of usage of many cores. Another fact is the wordlength utilized of 1.6 bits in average for weights and 1 for activation negatively impact into the final classification accuracy. The present work serves as an alternative, with higher classification accuracy and lower energy per classification. Higher frame rates could be increased by compromising area.

Chapter 6
Conclusion and Future Work

As we currently live in the renaissance of machine learning algorithms, the trade-off between specification and generalization leans towards higher flexibility and programmability. The application is in constant change, and no MLs architecture design seems to attend the aspiration of the consumers fully.

This scenario demands efficient programmable hardware capable of delivering the next best algorithm, sharing fundamental building blocks to its predecessor, and thus having enough design space for hardware gains. This work exploited this margin and benefited from it.

Results obtained by hardware accelerating a typical case of the most computationally expensive operation, the (5x5) convolution, yielded impressive results when compared to our reference software implementation. Four order magnitude acceleration was achieved, having an estimated speed-up of three times when in comparison to the processor alone for the MNIST case.

Our results strongly indicate that there may be no other computation bottleneck to be further accelerated. All other CNNs software implemented building blocks, including the sum of all convolutions in our reference model, individually take less than one percent of the new estimated computation time. As for a future project, other more demanding dataset or architectures could be used (as Inception V4 for ImageNet) in order to identify further computational bottlenecks not required by MNIST usage.

This project indicates that in order to achieve greater single image classification speed-up, increasing memory bandwidth, load/store units, and/or incrementing the wordlength might offer significantly final speed-up.

Furthermore, this project hopes also to propose that even if the number of load/store units were radically increased, one needs to consider that the amount of operations per convolutional layer is reduced by four, since a 2x2 no overlapping max-pool downsamples every four samples down to one each time it is performed. Thus one better not consider the worst case scenario when considering further optimizations.

As a future project, apart from co-simulating the hardware accelerator and the Xtensa processor, we would suggest rethinking the architecture, as the present work when scaled up would converge into a GPU-like architectures, with many processors and accelerators working in parallel. Finding out a possibility to lower the number of memory instructions, while scaling down the processor, avoiding unnecessary overhead, and using it just to control other CNNs building blocks

seems a possibility to us. Additionally a pure C optimized implementation of the building blocks could be done in order to minimize the processor's overhead.

References

[1] W. Van Dyke Bingham. *Aptitudes and aptitude testing*. Harper & brothers, 1937.

[2] R. J. Sternberg. *Handbook of Intelligence*. Ed. by S. Goldstein, D. Princiotta, and J. A. Naglieri. New York, NY: Springer New York, 2015. ISBN: 978-1-4939-1561-3. DOI: 10.1007/978-1-4939-1562-0. URL: http://link.springer.com/10.1007/978-1-4939-1562-0.

[3] D. Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017). ISSN: 0028-0836. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[4] H. A. Haenssle et al. "Man against machine: diagnostic performance of a deep learning convolutional neural network for dermoscopic melanoma recognition in comparison to 58 dermatologists". In: *Annals of Oncology* 29.8 (Aug. 2018), pp. 1836–1842. DOI: 10.1093/annonc/mdy166. URL: https://dx.doi.org/10.1093/annonc/mdy166.

[5] J. F. PUGET. *What is machine learning?* 2016. URL: https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Is_Machine_Learning?lang=en (visited on 01/23/2019).

[6] P. Mozur. *Googles AlphaGo Beats Chinese Go Master in Win for AI*. 2017. URL: https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html (visited on 01/23/2019).

[7] Peter Bright. *Elon Musk's Dota 2 AI beats the professionals at their own game*. 2017. URL: https://arstechnica.com/gaming/2017/08/ai-bot-takes-on-the-pros-at-dota-2-and-wins/ (visited on 01/23/2019).

[8] A. Rajeswaran et al. "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations". In: *CoRR* (Sept. 2017). arXiv: 1709.10087. URL: http://arxiv.org/abs/1709.10087.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[10] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.4.541.

[11] A. Waibel et al. "Phoneme Recognition Using Time-Delay Neural Networks". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* (1989). DOI: 10.1109/29.21701.

[12] K. Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (Mar. 1980), pp. 193–202. ISSN: 0340-1200. DOI: 10.1007/BF00344251.

[13] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521 (May 2015). ISSN: 0028-0836. DOI: 10.1038/nature14539. URL: http://www.nature.com/articles/nature14539.

[14] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep learning*. MIT Press, 2016. ISBN: 9780262035613. URL: https://www.deeplearningbook.org/.

[15] C. Olah. *Neural Networks, Manifolds, and Topology*. 2014. URL: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/ (visited on 01/23/2019).

[16] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* (Sept. 2015). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[18] Y. LeCun, C. Cortes, and C. J. Burges. *THE MNIST database of handwritten digits, Yann LeCun, Corinna Cortes and Chris Burges*. 1998. arXiv: /repositorio.ug.edu.ec/handle/redug/10118 [http:]. URL: http://yann.lecun.com/exdb/mnist/ (visited on 01/24/2019).

[19] Xing Han Lu. *Dash t-SNE Explorer*. URL: https://dash-gallery.plotly.host/dash-tsne (visited on 01/24/2019).

[20] D. Cireşan, U. Meier, and J. Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". In: *CoRR* (Feb. 2012). arXiv: 1202.2745. URL: http://arxiv.org/abs/1202.2745.

[21] Benenson Rodrigo. *Classification datasets results*. 2016. URL: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html (visited on 01/23/2019).

[22] L. Wan et al. "Regularization of Neural Networks using DropConnect". In: *Proceedings of the 30th International Conference on Machine Learning*. PMLR, 2013, pp. 1058–1066.

[23] I. Sato, H. Nishimura, and K. Yokoi. "APAC: Augmented PAttern Classification with Neural Networks". In: *CoRR* (May 2015). arXiv: 1505.03229. URL: http://arxiv.org/abs/1505.03229.

[24] J.-R. Chang and Y.-S. Chen. "Batch-normalized Maxout Network in Network". In: *CoRR* (Nov. 2015). ISSN: 662243. arXiv: 1511.02583. URL: http://arxiv.org/abs/1511.02583.

[25] C.-Y. Lee, P. W. Gallagher, and Z. Tu. "Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree". In: (2015). arXiv: 1509.08985. URL: http://arxiv.org/abs/1509.08985.

[26] Z. Liao and G. Carneiro. "On the Importance of Normalisation Layers in Deep Learning with Piecewise Linear Activation Units". In: *CoRR* (2015). DOI: 10.1109/WACV.2016.7477624. arXiv: 1508.00330. URL: http://arxiv.org/abs/1508.00330.

[27] B. Graham. "Fractional Max-Pooling". In: *CoRR* (2015). arXiv: 1412.6071. URL: http://arxiv.org/abs/1412.6071.

[28] Z. Liao and G. Carneiro. "Competitive Multi-scale Convolution". In: *CoRR* (Nov. 2015). arXiv: 1511.05635. URL: http://arxiv.org/abs/1511.05635.

[29] D. C. Ciresan et al. "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition". In: *Neural Computation* (2010), pp. 3207–3220. DOI: 10.1162/NECO_a_00052. arXiv: 1003.0358. URL: http://arxiv.org/abs/1003.0358.

[30] C. Szegedy et al. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: *CoRR* (Feb. 2016). DOI: abs/1602.07261. arXiv: 1602.07261. URL: http://arxiv.org/abs/1602.07261.

[31] Cadence. *Tensilica Xtensa LX7 Data Sheet*. Tech. rep. Cadence Design Systems, 2016. URL: https://ip.cadence.com/uploads/1099/TIP_PB_Xtensa_lx7_FINAL-pdf (visited on 02/17/2019).

[32] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach, 5th edition*. 2011, p. 857. ISBN: 9780123838728.

[33] Cadence. *Whitepaper: TIE Language-The Fast Path to High-Performance Embedded SoC Processing*. 2016. URL: https://ip.cadence.com/uploads/980/TIP_WP_TIE_FINAL-pdf (visited on 02/17/2019).

[34] A. Traber and M. Gautschi. *PULPino: Datasheet*. Tech. rep. 2017. URL: https://www.pulp-platform.org/docs/pulpino_datasheet.pdf (visited on 02/17/2019).

[35] K.K.Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999. ISBN: 9788126510986.

[36] D. Bankman et al. "13.5 13.5 An Always-On 3.8μJ/86% CIFAR-10 Mixed-Signal Binary CNN Processor with All Memory on Chip in 28nm CMOS". In: *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*. 2018, pp. 222–224. DOI: 10.1109/ISSCC.2018.8310264.

[37] S. K. Esser et al. "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing". In: *CoRR* (2016). DOI: 10.1073/pnas.1604850113. arXiv: 1603.08270.

[38] K. Ando et al. "BRein memory: A 13-layer 4.2 K neuron/0.8 M synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm CMOS". In: *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*. 2017, pp. C24–C25. ISBN: 9784863486065. DOI: 10.23919/VLSIC.2017.8008533. arXiv: 1602.02830.

[39] F. N. Buhler et al. "A 3.43TOPS/W 48.9pJ/pixel 50.1nJ/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm CMOS". In: *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*. 2017, pp. C30–C31. ISBN: 9784863486065. DOI: 10.23919/VLSIC.2017.8008536.

Printed by Tryckeriet i E-huset, Lund 2019

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-686
http://www.eit.lth.se