NTNU
Norwegian University of
Science and Technology

TDT4506 Specialisation project

Convolutional Neural Networks and their Potential Hardware Acceleration

Magnus Halvorsen

Fall 2014

Department of Computer and Information Science
Norwegian University of Science and Technology

Supervisor 1: Donn Alexander Morrison
Supervisor 2: Yaman Umuroglu

Assignment

Candidate name: Magnus Halvorsen

Assignment title: Energy efficient machine learning algorithms in hardware

Supervisors: Donn Alexander Morrison and Yaman Umuroglu

Assignment text:
The aim of this project is to explore the implementation of machine learning algorithms in hardware (e.g., FPGA) with the intention of improving energy efficiency over traditional implementation in software. The algorithm should be modularly developed so that it can be potentially used as an accelerator tile within a multi-core heterogeneous computing platform such as SHMAC.

For a given algorithm, e.g., a deep neural network, the student will:

1. Explore the feasibility and expected efficiency gain over a general-purpose CPU (e.g., is there a need?)
2. A literature survey on related techniques, hardware implementations, etc.
3. Allocation of components between hardware and software (e.g., according to a HW/SW co-design methodology)
4. If time permits, adapting the module for use as a SHMAC accelerator tile

Abstract

The breakdown of Moore's Law and the effects of Dark Silicon have forced a paradigm shift from performance-centric serial computation to energy-efficient parallel computation. This has sparked an interest in heterogeneous computing, where the processor contains different cores and accelerators that are specialized for certain tasks. The SHMAC project aims to provide a research platform for heterogeneous systems research. This has led to a need of exploring what kind of applications are worth accelerating.

One such application is the Convolutional Neural Network algorithm, which is a state of the art technique for recognizing objects in images and sound. In this report we will explore the potential gains of creating a hardware accelerator for such a network. We will introduce the mathematical fundamentals behind the algorithm, provide an overview of the most recent suggested hardware architectures, and present our architecture, Imagezor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

During the last decade it has become apparent that *Moore's law* is coming to an end, due to the breakdown of *Dennard scaling*. This has forced a paradigm shift in the computing domain, from performance-centric serial computation to energy-efficient parallel computation [**?**]. This comes from the observation that many simpler and smaller parallel processors generally provides better energy-efficiency than a single high-performing superscalar one. Unfortunately this is not enough. The continued decline of Moore's law has led to a *utilization wall* [**?**], which causes the percentage of a chip one can actively use within a chip's power budget to drop exponentially. Thus only a portion of the transistors on a chip can be powered at same time for a given power budget and a constant die area. This is referred to as the *Dark Silicon effect*.

In order to reduce the effect of Dark Silicon there have been purposed two main strategies [**?**]. First, mutli-core architectures can be made heterogeneous by including cores that represent different points in the power/performance design space. Thus the architecture must be able to decide which core should perform the computation based whether the goal is performance or energy efficiency, or a combination of both. The second is using hardware components that are specialized for a specific task, and thus does it preferably both fast and energy-efficient. If the tasks does not occur, the component can be turned off, preventing it from wasting energy. Such specialized components are known as *hardware accelerators*.

The Single-ISA Heterogeneous MAny-core Computer(SHMAC) [**?**] project is an ongoing project at the Energy Efficient Computing Systems(EECS) research area at the Norwegian University of Science and Technology. The purpose of the project is to mitigate the Dark Silicon effect by using the techniques described in the previous paragraph. There have thus been an increasing interest in exploring what kind of applications are worth hardware-accelerating.

One of the applications that have been considered are *machine learning algo-*

rithms. A concept that is several decades old, but has been increasingly popular in recent years. There are two primary reasons for this. First, many machine learning algorithms require a great amount of data for learning, in order to make accurate predictions or detect patterns. This has been largely solved by the vast amount of data that has become available due to *Big Data* and the Internet. Second, most machine learning algorithms are computationally expensive, making many of them previously infeasible to make use of. But with the advancement of processing power and sophistication of processor architectures, the execution time has come to a reasonable level. Especially the arrival of the *Graphic Processcing Unit* (GPU) helped a lot, since they are able to exploit the parallel nature of many machine learning algorithms.

An machine learning algorithm that have become recently popular is the *Convolution Neural Network* (CNN), which is an extension of the *Artifical Neural Network* (ANN) algorithm. It is one of the state of the art techniques used for object recognition in images and sound. A technique that is used by Google and Facebook for face and object detection in their image databases. With the Internet-of-Things and an increasing amount of devices able to take pictures and film, the potential for CNNs have vastly increased. By making our devices able to recognize its surroundings one can create various interesting applications.

For the reasons mentioned above, we have in this project decided to explore the feasibility of hardware accelerating a Convolutional Neural Network. In this report we will give an introduction to the mathematical model behind Artificial Neural Networks and Convolution Neural Network. We will give an overview of the latest research done on hardware accelerating such networks, and purpose a potential design that can be used on a Spartan 6 FPGA. Finally we will review the performance and energy-efficiency gain from this design.

1.2 Assignment Interpretation

Based on the assignment description text, the following main tasks were identified:

Task 1 ***(mandatory)*** Choose a machine learning algorithm to investigate.

Task 2 ***(mandatory)*** Determine if a hardware accelerator will provide significant performance and energy-efficiency gains.

Task 3 ***(mandatory)*** Begin the development of a hardware accelerator for the chosen machine learning algorithm.

Task 4 ***(mandatory)*** Provide an overview of the state of the art of software and hardware implementations of CNNs.

Task 5 ***(optional)*** Adapt the module to a SHMAC accelerator tile.

We wish to note that task 1 goes beyond simply choosing an algorithm. Since the student had no background within artificial intelligence, a lot of effort have gone to learning and understanding the given algorithm.

1.3 Report structure

For the convinience of the reader, we will here provide a quick overview of the topic of the report's chapters.

Chapter 2: Background gives an introduction to the mathematical model of Artifical Neural Networks and Convolutional Neural Networks.

Chapter 3: Related Work gives an overview of the state of the art CNNs, and the most relevant recent hardware implementations.

Chapter 4: Architecture presents our suggested design for a hardware accelerator for a CNN.

Chapter 5: Results and Discussion compares our design with a equivalent implementation on a CPU, with respect to performance and energy efficiency. The chapter will also give our thought on the given results.

Chapter 6: Future Work presents how the design can be further improved.

Chapter 7: Conclusion provides concluding remarks and a summary of which tasks we were able to complete.

Chapter 2

Background

In this chapter we will go through the fundamental mathematics and concepts behind the *Convolutional Neural Network* (CNN) model. It gives a basic introduction to both general neural networks and *CNNs*.

2.1 Artificial Neural Networks

An *Artificial Neural Network* (ANN) [**?**][**?**] is a computational model that is used for machine learning and pattern recognition. The name and basic concept is inspired by how the animal brain uses a network of neurons to recognize and classify objects. Depending on the input different neurons *activate* (or *fire*), making the brain able to decide what kind of pattern it is detecting.

An ANN can intuitively be viewed as a probabilistic classifier. Depending on the input data it will calculate the probability that the data belongs to a certain *class* (e.g. an object in an image or an investment decision). The network can be trained to recognize different classes by being provided a set of labeled training data, e.g. a set of faces and a set of non-faces. It can then learn to decide whether an image contains a face or not. This is called supervised learning. The network can also be trained unsupervised, by providing it with a set of unlabeled images. The latter technique is used to find hidden structures in the data, by learning the network to recreate the input. But for this project only supervised learning is relevant.

2.1.1 Definition

An ANN consists of a number of layers containing a set of so-called *neurons* (see Figure 2.1), also known as *units*. A neuron takes in a set of values as input (e.g. image pixels), where each value is associated with a respective weight. The input and the weights are multiplied and summed, and the result is used to

calculate a non-linear *activation function*. Formally a neurons input and output is defined as:

$$Input : \{x_1, x_2, \ldots, x_n\} = \mathbf{x} \tag{2.1}$$

$$Output : f(\mathbf{w^T x}) = f(\sum_{i=1}^{n} w_i x_i + b) = o \tag{2.2}$$

Where $\mathbf{w}$ is the vector containing connection weights and b is the neuron bias. *f(...)* is the activation function, which eumulates the activation of a neuron in the brain, i.e. it decides whether the neuron is on or off. It also causes the values in the network to have a reasonable value interval. *f(...)* tends to be either:

$$Sigmoid : f(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

$$\text{Hyperbolic tagent:} f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.4}$$

The reason these functions are used is that they have the non-linear property, which increases the expressivness of the network. Thus reducing the number of neurons the network needs to solve a given problem. In addition both function have ranges [0, 1] and [-1, 1], respectively, which translates well into probability computation. I.e. you can view the value of the activation function as the probability of that neuron activating.

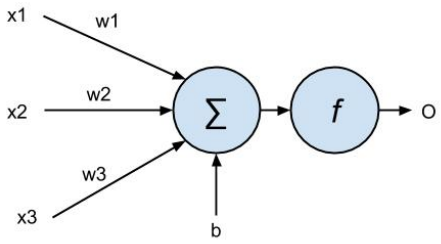

Figure 2.1: A single neuron with three inputs.

An ANN consist of n_l layers, each containing a set of neurons. The first layer is *the input layer*, and the last layer is *the output layer*. The layers in between are called *the hidden layers*. Each layer uses the previous layer's output as input. The input layer is provided with the initial input and uses it to calculate the activation function for each of its neurons. The result is propagated to the first hidden layer, and continues up until it reaches the output layer, which provides the final output. This is known as a *feedforward neural network*.

The network takes in two parameters:

$$(\mathbf{W}, \mathbf{b}) = (\mathbf{W}^{(1)}, b^{(1)}, \mathbf{W}^{(2)}, b^{(2)}, \ldots, \mathbf{W}^{(n_l)}, b^{(n_l)}) \tag{2.5}$$

Where $\mathbf{W}$ is a 3-dimensional matrix containing the weight matrix for each layer. $\mathbf{W}^{(l)}$ contains the weight matrix for the weights going from layer l to $l+1$. E.g., in the case of Figure 2.2, $\mathbf{W}^{(1)} \in \Re^{3\times 4}$ and $\mathbf{W}^{(2)} \in \Re^{4\times 2}$.

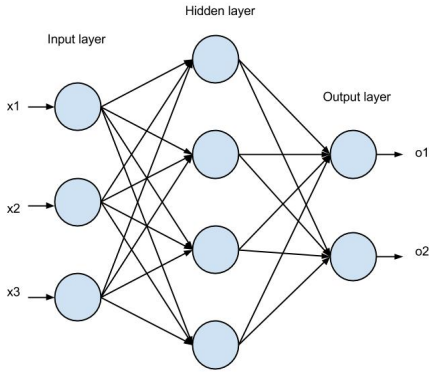

Figure 2.2: An Artificial Neural Network.

2.1.2 Training

During the training of the network it is the parameters $(\mathbf{W}, \mathbf{b})$ that are altered in order to adapt the network to the training data. This is done by providing the network with a set of training samples, where we provide an input and an expected output. By using a cost function we can then figure out how we should tune our weights and biases in order to reduce the error rate. In other words, our goal is to minimize a cost function over a set of training samples. This can be done by using *gradient descent* and the *backpropagation algorithm* [**?**][**?**][**?**].

Let the cost function for a single training example (x, y) be defined as:

$$Cost(\mathbf{W}, \mathbf{b}; x, y) = \frac{1}{2}(h_{\mathbf{W},\mathbf{b}}(x) - y)^2 \tag{2.6}$$

Where x is the input, $h_{\mathbf{w},b}(x)$ is the actual output of our network and y is the correct output. Then the cost function for m training examples $((x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m))$ is:

$$Cost(\mathbf{W}, \mathbf{b}) = \frac{1}{m}\sum_{i=1}^{m} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\mathbf{w}_{ji}^l)^2 \tag{2.7}$$

Where the first term is simply the average sum-of-squares error. The second term is the *regularization term*, or *weight decay term*, which tends to reduce

overfitting. ANNs have a vast number of parameters, i.e. weights, which makes it susceptible to random noise. This can greatly reduce the networks ability to provide correct predictions, but this can be mended by the regularization term.

Based on this we can use gradient descent to compute how we should alter the weights in order to reduce the cost function. One iteration of gradient descent updates $\mathbf{w}$ and b as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial}{\partial w_{ij}^{(l)}} Cost(\mathbf{W}, \mathbf{b}) \tag{2.8}$$

$$b^{(l)} = b^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} Cost(\mathbf{W}, \mathbf{b}) \tag{2.9}$$

Where α is the learning rate, which is a predetermined constant. w_{ij}^{l} denotes the weight between neuron j in layer l, and neuron i in layer l+1. b_i^l denotes the bias associated with neuron i in layer l+1.

Note that this would only make us able to compute the gradient for the output layer. In order to perform gradient descent on the hidden layers, we need to propagate the error from the output layer backwards, to the hidden layers. For this we use the *backpropagation algorithm*. Let $o_i^{(l)}$ denote the output of the ith neuron in layer l, and $z_k^{(l)}$ is the weighted sum of the inputs plus the bias for the kth neuron in layer l. Then the *backpropagation algorithm* can be formalized as follows:

1. Perform a feedforward pass, computing the output of every layer.

2. For each output neuron k in the output layer, compute *the error term*:

$$\delta_k = \frac{\partial}{\partial z_k^{(n_l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = -o_k^{n_l}(1 - o_k^{n_l})(y_k - o_k^{n_l}) \tag{2.10}$$

3. For each hidden layer $l = n_l - 1, n_l - 2, \ldots, 2$ compute:

$$\delta_i^l = o_i^l(1 - o_i^l) \sum_{j=1}^{s_{l+1}} w_{ij}^l \delta_j^{l+1} \tag{2.11}$$

4. Compute the partial derivative for each weight and bias:

$$\frac{\partial}{\partial w_{ij}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = o_j^{(l)} \delta_i^{(l+1)} \tag{2.12}$$

$$\frac{\partial}{\partial b^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \delta_i^{(l+1)} \tag{2.13}$$

Now, combining *gradient descent* and the *backpropagation algorithm* we can describe an algorithm to train our network:

1. Initialize the weights $\mathbf{w}^{(l)}$ and b^l to random values for every layer l.
2. Do steps 3 to 5 until the $Cost(\mathbf{W}, \mathbf{b})$ function is low enough or converges. This is referred to as an *epoch*.
3. Set $\Delta\mathbf{w}^{(l)} := 0$ and $\Delta b^{(l)} := 0$ for all l.
4. For i = 1 to m,
   (a) Use the backpropagation algorithm to compute $\nabla_{\mathbf{w}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$ and $\nabla_b^{(l)} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$ for every layer l.
   (b) Set $\Delta\mathbf{w}^{(l)} := \Delta\mathbf{w}^{(l)} + \nabla_{\mathbf{w}^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$.
   (c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x^{(i)}, y^{(i)})$.
5. Update the parameters:

$$\mathbf{w}^{(l)} = \mathbf{w}^{(l)} - \alpha[(\frac{1}{m}\Delta\mathbf{w}^{(l)}) + \lambda\mathbf{w}^{(l)}]$$

$$b^{(l)} = b^{(l)} - \alpha[\frac{1}{m}\Delta b^{(l)}]$$

2.1.3 Issues with object recognition

While the ANN model have proven useful in several applications, it falls short when it comes to object recognition in images. According to [**?**] there are three major reasons for this: .

1. **Topology**. A fully connected ANN does not take into consideration the topology of the input. An image has a strong 2D spatial locality correlation, which makes it possible to combine low-order features (edges, end-points etc.) in the same area into higher-order features.
2. **Scalability**. Even small images contains a large amount of pixels/inputs, e.g. a 32×32 image contains 1024 pixels/inputs. A fully connected network with 100 hidden units would then end up with 1024×100 weights that needs to be calculated in the first layer. Thus making it harder to scale for larger images and rather inefficient .
3. **Object variance.**While objects are similar enough, on a higher level, to be grouped together into a class, they can still be very different on a lower level. E.g. a human face have several features that are needed for it to be defined as a face, e.g. eyes, mouth, nose etc. But the size and shape of these features tend to be very different from person to person. While it is possible for a standard ANN to compensate for these internal differences within a class, it would have to make three costly compensations. 1) The network would have to be very large, 2) it would probably contain several neurons with similar weight vectors positioned at different places in the network, and 3) it would require a massive amount of training samples.

2.2 Convolutional Neural Network

A *Convolutional Neural Network* [**?**] (CNN) is an extension of the *Artificial Neural Network* model, which is made specifically for object recognition in images or speech recognition. It was made in order to solve the issues that the classic ANN model faced.

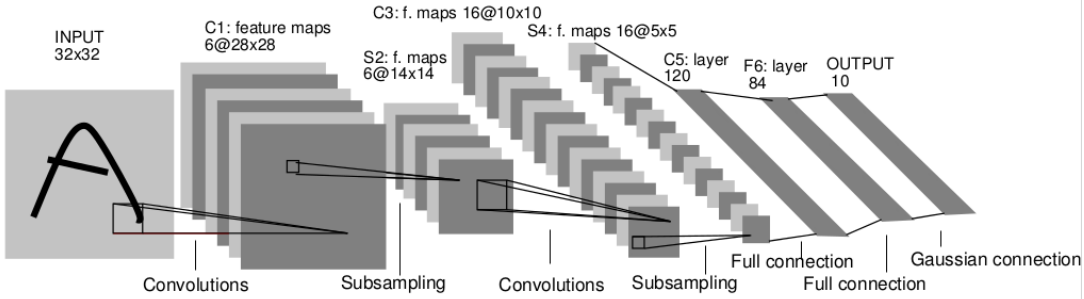

Figure 2.3: An example CNN, the LeNet-5 [**?**]. It consists of two convolution and subsampling/pooling layer pairs, which are connected to a fully connected ANN with 10 output classes.

2.2.1 Definition

The CNN model adds two additional types of layers, in addition to the standard ANN layers: a *convolution layer* and a *subsampling/pooling layer*. The idea behind the two new layers is to exploit the local 2D structure of images, i.e. pixels close to each other are highly correlated. By using local correlation one can extract and combine small local features (e.g. edges, corners, points) into higher-order features (e.g. a nose, a mouth, a forehead), which can in the end be recognized as an object (e.g. a face). A full network is illustrated in Figure 2.3.

Convolution layer
The convolution layer extracts a set of features from a set of input images. For each feature, the respective feature is extracted from all the input images and put in a feature map. E.g. if the filter extracts vertical edges, only the vertical edges from all the input images would remain in the resulting feature map. Thus different features can be extracted by having several feature maps that extracts different features.

The extraction is done by performing a *convolution operation* on the image, using a *kernel* that acts like a filter. The kernel is a 2D matrix that contains a set of weights. Depending on values of the weights, convoluting the image with the kernel will have wide range of effects, e.g. sharpening, bluring, edge

detection and feature extraction. By training our network we can configure the weights to extract the features we need in order to recognize our desired classes.

After the convolution operation has been performed, a bias is added to every element in the feature map and the result is sent through a non-linear function, e.g. a sigmoid function.

Formally we can define the convolution layer as follows. The layer accepts n images $X_1, X_2, \ldots, X_n$ as inputs, and produces m feature maps, $F_1, F_2, \ldots, F_m$. These feature maps are produced using a set of m learned kernels $W_1, W_2, \ldots, W_m$. Each feature map F_t is then produced by computing:

$$\mathbf{F}_t = Sigmoid(b + \sum_{i=1}^{n} \mathbf{X}_i * \mathbf{W_t}) \tag{2.14}$$

Where $\mathbf{F}$ is the resulting feature map, X is the input image, $\mathbf{W}$ is the kernel matrix, and b is the bias. $X * W$ is the convolution operation, which is defined as:

$$y_{ij} = \sum_{q=1}^{k} \sum_{p=1}^{k} x_{i+q,j+p} w_{qp} \tag{2.15}$$

Where x_{ij} is a value of the input matrix, w_{mn} is a value in the $k \times k$ kernel matrix, and y_{ij} is a value of the output matrix.

E.g. consider the LeNet-5 in Figure 2.3, in the first layer C1 the input is a single 32×32 image which is convoluted with 6 kernels, producing 6 feature maps. Thus $n = 1$ and $m = 6$. The resulting feature maps are then further processed by a subsampling/pooling layer S2 (see next section), which are used as input to the next convolutional layer C3. The six processed feature maps are then convoluted with 16 kernels, producing 16 new feature maps. Thus in this layer $n = 6$ and $m = 16$.

This helps solve the first two issues from Section 2.1.3. The neurons in a feature map share the same kernel, thus the same weights, which greatly reduces the size of the network. The convolution operation applies a 2D filter on the image, which makes the network able to exploit the spatial correlation in the image.

Subsampling/pooling layer

Once a feature has been detected, the exact position become less important. For example, the distance between the mouth and the eyes tend to vary between persons. So in order to make the CNN not too sensitive to the relative placement of features, the accuracy of the all feature maps needs to be reduced. This can be done by subsampling (i.e. partitioning) the feature map into $s \times s$ non-overlapping submatrices, and then perform a pooling operation on each respective matrix. There are two types of pooling operations which are used for CNNs:

- *Max-pooling* extracts the maximum value of the submatrix.

- *Average-pooling* extracts the average value of all the elements in the sub-matrix.

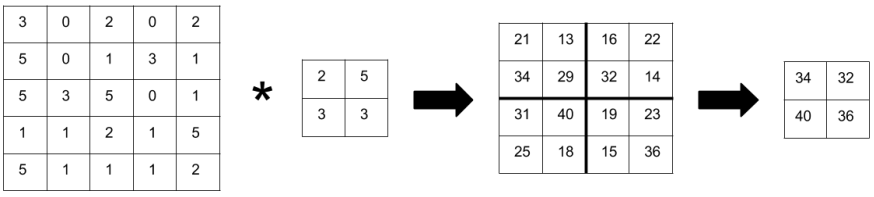

Figure 2.4: Illustration of the convolution and subsampling/max-pooling operations. The leftmost matrix is convulted with a 2×2 kernel, and the resulting matrix is subsampled into four non-overlapping areas where the max value is extracted.

We can then formally define subsampling/pooling as:

$$\mathbf{O} = subsample\_pool(\mathbf{F}) \tag{2.16}$$

Where $\mathbf{O}$ is the output matrix, $\mathbf{F}$ is the input feature map, and the *subsample_pool()* function's operation is defined as either:

$$o_{ij} = max(x_{i \times s+p, j \times s+q}) \qquad q, p \in 1, 2, \ldots, s \tag{2.17}$$

or

$$o_{ij} = \frac{1}{s^2} \sum_{p=1}^{s} \sum_{q=1}^{s} x_{i+p, j+q} \tag{2.18}$$

Where o_{ij} is a value in the output matrix and f_{ij} is a value in the feature map, and s is the dimension of the subsampling size. A max-pooling operation is illustrated in Figure 2.4.

Thus, the subsample/pooling layer helps solve the two last issues from Section 2.1.3. By reducing the accuracy, the network is less sensitive to the difference between instances of a class. This also causes the network size to be smaller, since it does not require neurons to recognize the differences.

Figure 2.4 illustrates the convolution operation and the subsample/max-pooling operation, while Figure 2.5 illustrates the full operation of the convolution and subsampling/pooling layers.

2.2.2 Training

As mentioned, a CNN consists of three types of layers: a convolution layer, a subsampling/pooling layer and fully connected layer. The latter is trained as described in Section 2.1.2, using backpropagation and gradient descent. The two

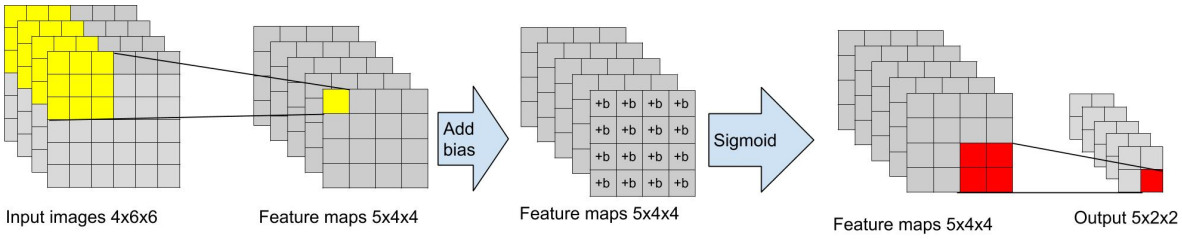

Figure 2.5: A visual overview of the operations performed by the convolution layer and subsampling/pooling layer with four input images. Yellow represents the convolution operation, and red represents the subsampling/max-pooling operation.

other layers use the same general algorithm, but the error δ^l and the gradient of $Cost(\mathbf{W}, \mathbf{b}; x, y)$ is calculated differently.

Since the backpropagation aglorithm starts at the last layer and work its way backwards, the error is first calculated for the fully connected layers. It is then provided to the subsampling/pooling layer, and finally to the convolution layer. Thus we first need to calculate the error for the subsampling/pooling layer, so we can propagate it to the convolution layer.

The subsampling/pooling layer does not contain any weights, and can therefore not be tuned. Thus it only needs to propagate the error it receives. Depending on which pooling operation is used, there are two respective methods for this. For max pooling, the error is simply propagated to the neuron that was chosen as the maximum value, while the rest are set to zero. For average-pooling we have to distribute the error evenly between all the responsible neurons. We therefore define the function *upsample(...)*, which performs the correct propagation operation depending on the type of pooler.

We can now formally define how to calculate the error and the gradient by simply replacing the equations in step 3 and 4 in the backpropagation algorithm with the following equations. For simplicity we assume that convolution and subsampling/pooling is done in a single layer l.

$$\delta_k^l = upsample((\mathbf{W}_k^l)^T \delta_k^{l+1}) \bullet f'(\mathbf{Z}_k^l) \tag{2.19}$$

Where $(\mathbf{W}_k^l)^T$ is the weight matrix in layer l, δ_j^{l+1} is the error matrix for layer $l+1$, $\bullet$ is the element-wise product (i.e. Hadamard product), $f'(\mathbf{Z}_k^l)$ is the matrix containing the derivative of the activation function, and k indexes the filter number. I.e. it contains $o_{kij}^l(1 - o_{kij}^l)$ for every neuron at index ij in feature map k in layer l.

Using this we can calculate the gradient:

$$\frac{\partial}{\partial \mathbf{w}_k^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \sum_{i=1}^{m} (\mathbf{o}_i^{(l)}) * \delta_k^{(l+1)} \tag{2.20}$$

$$\frac{\partial}{\partial b_k^{(l)}} Cost(\mathbf{W}, \mathbf{b}; x, y) = \sum \delta_k^{(l+1)} \tag{2.21}$$

2.3 Potential for parallelism

A vast amount of the computation required by a CNN can be parallelized. Thus, in order to accelerate the processing of the network it is important that these potential parallelizations are identified and exploited. The most obvious being:

1. The computation of each of the individual feature maps and their corresponding subsampling/pooling. Which [**?**] referred to as *inter-parallelism*.
2. It is also possible to parallelize the computation of the feature maps that take more than one matrix as input. This is the case in the subsequent layers after the first. Which [**?**] referred to as *intra-parallelism*.
3. The activation of each neuron in the fully connected layer.

One can also parallelize the mentioned parallelizations themselves. The convolution of a matrix $n \times n$ using a $k \times k$ kernel consists of $(n-k+1) \times (n-k+1)$ convolution operation, which each can be done in parallel. Thus convoluting the whole matrix could potentially take only the time it takes to perform one convolution operation. The subsampling/pooling operation can also be parallelized by pooling all of the individual submatrices at the same time.

It is also possible to compute the neuron activation function in parallel, since its a series of n multiply and accumulate operations. One option is to parallelize them by creating a binary tree multiplier, where you have n units compute the product of the input and its respective weight, then you use $\frac{n}{2}$ units to add two of the results each, and so on until you have a single value. This will reduce the time it takes from n time to $log_2 n$ time. But considering the vast amount of activation functions that are to be computed, hardware resource are better spent on computing as many activation functions in parallel as possible. Unless you have an unlimited amount of hardware resources.

2.4 Hardware Acceleration

We chose to implement a variant of the LetNet-5 CNN for digit recognition using *the MNIST dataset*. The MNIST dataset consist of 50 000 samples of handwritten digits ranging from 0-9, where 40 000 of the samples are used for training and 10 000 samples are used to determine the accuracy of the network. The reason for choosing the LetNet-5 network is mostly due to it being a fairly small and simple network, making it eaiser to implement and debug. In addition it has been shown to work very well on the MNIST dataset. This is important since the aim of this project is hardware acceleration of Convolutional Neural Networks, not configuring a network with the optimal topology for a specfic

task. Thus using a network that is already configured for a dataset gives us more time to explore the hardware part.

We used XXXXX's framework for neural networks for the pure software implementation of the LeNet-5, which is available at XXXXX. For a detailed introduction of the LeNet-5 network, please refer to the orignal paper [**?**]. We later extended the framework to include our hardware accelerator, this is described in more detail in section

2.5 XXXXXX

.

Chapter 3

Related Work

This section will give an overview of the current state of research on Convolutional Neural Networks.

3.1 Convolutional Neural Networks

The mathematical fundamentals for Convolutional Neural Networks was introduced as early as in the 1980s by Kunihiko Fukushima[**?**][**?**], in form of the *neocognitron* model. The model was later improved in 1998 by Yann LeCun, Lon Bottou, Yoshua Bengio, and Patrick Haffner - who introduced the *Convolutional Neural Network* model. In 2003 the model was simplified by Patrice Simard, David Steinkraus, and John C. Platt [**?**], in an attempt to make it easier to implement. The paper also mentions two of the main issues with CNNs: the size of the training set and the time spent training. In order to achieve high enough accuracy a CNN requires thousands of training samples, which needs to be labeled. Processing all of these samples and fine-tuning the networks takes a great amount of processing power, causing training to take days or weeks. These issues are the ones that caused CNNs not to gain popularity before mid-2000. The rise of the Internet, digital cameras, and Big Data have provided us with vast amounts of images which can be used for training. Improvements in the speed and sophistication of computer hardware have reduced the training time from days/weeks to hours. E.g. [**?**] purposes a GPU implementations which reduced the epoch (see Section 2.1.2) training time from 35 hours to 35 minutes. This demonstrates that highly parallel hardware vastly increases the effiency of neural networks compared to CPUs.

These recent advancements have renewed the interest in neural networks, and increased the research done on the field. As a result CNNs have become a leading model within pattern recognition for computer vision. This can be illustrated by the fact that CNNs implementations have won several pattern recognition contests in the period 2009-2012, including IJCNN 2011 Traffic Sign Recognition Competition[**?**] and the ISBI 2012 Segmentation of Neuronal

Structures in Electron Microscopy Stacks challenge[**?**].

3.2 Convolutional Neural Network in Hardware

There have been several proposed hardware architectures during the last decade, including [**?**][**?**][**?**][**?**][**?**]. Below we will describe some of the more recent and relevant architectures.

In [**?**] a CNN was implemented on a Virtex-4 SX35 FPGA from Xilinx. In this implementation all the fundamental computations were hard-wired, and controlled by a 32 bit soft processor using macro instructions. Training was done offline, and a representation of the network was provided to the soft processor. With this implementation they were able to process a 512×384 grayscale image in $100ms$, i.e. 10 frames per second. The design was intented for use in low power embedded vision systems, e.g. robots, and the whole curcuit board used less than 15 W.

Farabet and LeCun later improved the mentioned architecture in [**?**]. In this design they added multiple parallel vector processing units and allowed individual streams of data to operate seamlessly within processing blocks. They were able to achieve 30 frames per second using 15 W. In addition they predicted a planned ASIC implementation of the system would increase the processing speed and reduce the power to 1 W.

In [**?**] they explore how they can exploit the different the parallel nature of CNNs. They introduce types of parallelism found in CNNs, *inter-output* and *intra-output*. The first one comes from the observation that each feature map and the corresponding subsampling/pooling computation can be done in parallel. This is easily seen in the first layer. The second one refers to that the convolution of several inputs are combined to produce one feature map (see Figure 2.5), where the individual convolutions can be done in parallel. This one is present in all of the convolution layers after the first layer. They exploit these observations by purposing a dynamically configurable coprocessor on a FPGA, which can switch between computing several different feature maps in parallel and processing several inputs to compute one feature map. By doing this they are able to fully utilize the parallel nature of a CNN and reduce the intermediate storage on the FPGA. Using a Virtex 5 SX240T FPGA with 1024 multiply-accumulate units they were able to outperform a 2.3 GHz quad-core, dual socket Intel Xeon, and a 1.35 GHz C870 GPU by 4x to 8x.

[**?**] presents an architecture they named the *nn-X*. For the implementation they used a Xilinx ZC706 platform, containing a Kintex-7 FPGA and two ARM Cortex-A9. They made acceleration units for the convolution and subsampling/pooling layers on the FPGA, while the fully-connected layers was processed by the ARM processors. This implementation seems to be the fastest of all the purposed FPGA implementations to date, able to perform up to 227 G-ops/s. In addition it seems to be the most energy efficient across all platforms, by being able to perform 25 G-ops/s-Watt. In the paper they compared it to a Intel i7-3720QM and a NVIDIAs GTX 780 GPU, where it was shown that

it was up to 20x more power efficient. This makes a strong case for hardware acceleration of CNNs in applications designed for energy efficiency.

Chapter 4

Architecture

The aim of this project was to investigate the possibility of creating a hardware accelerator for a machine learning algorithm. We decided implement a module that could support the *the LeNet-5* [**?**] (see Figure 2.3). It was chosen for the reason that it is one of the easier CNN architectures to implement, and the module is only intended as a proof of concept. The input image is relatively small compared to the other architectures mentioned in Chapter 3, which greatly reduces the amount of intermediate storage needed on chip. In addition the size of the kernels are the same in both convolution layers, making us able to reuse the convolution hardware accelerator for both layers, without adding configuration logic for different kernel sizes.

4.1 Imagezor's Architecture

Imagezor takes n images as input, $I_1, I_2, \ldots, I_n$, a single kernel K, and outputs a processed image O. Using the input images and the kernel it performs the operations of the convolution and subsampling/pooling layer for a single kernel. Thus the output O is a subsampled/pooled feature map that has been produced by convolution the images $I_1, I_2, \ldots, I_n$ with the kernel K.

Imagezor can compute the convolution and subsample/pooling layer by doing the above computations for all the kernels in the respective layer. Thus one can exploit inter-parallelism by making several instances of the Imagezor architecture run in parallel. One can also exploit intra-parallelism, but then one need to connect the different Imagezor instances so they can add up the results from the convolutions without using the intermediate convolution buffer, as described in [**?**].

Imagezors pipeline consists of five major components (Figure 4.1):

- **The convoluter**. Performs the convolution operation on the input.
- **The intermediate convolution buffer**. Since the resulting feature map is the sum of the convolutions of all the input images (with the exception of

the first layer), this buffer is needed to store the results from the previous convolution, so that it can be accumulated with the current convolution. In the first layer of the network there is only one input image (i.e. $n = 1$), thus no summation is needed.

- **Sigmoid**. Performs the non-linear sigmoid function on the feature maps.
- **Subsample/max-pooler**. Performs the subsample/max-pool operation on the feature maps.

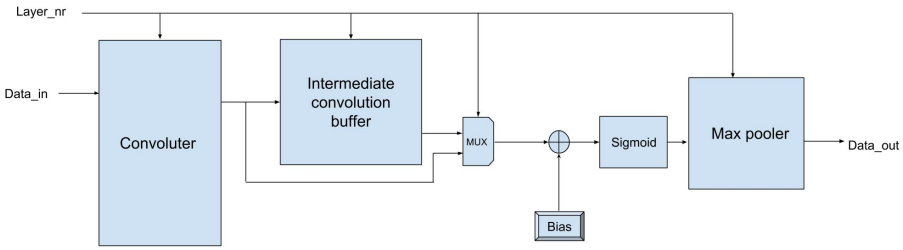

Figure 4.1: The architecture of Imagezor.

The *layer_nr* signal is used to define if it is the first or the second layer that is being computed. The input image in the first layer is bigger than in the second (32×32 vs 14×14), which the convoluter and the max pooler need to compensate for (see Section 4.1.1 and **??**). In addition in the second layer the intermediate convolution buffer needs to be activated so it accumulate and store all the convolutions needed to compute a single feature map. Thus the mux is needed to propagate the data directly from the convoluter in the first layer, and from the buffer in the second layer.

In order to reduce resources spent and execution time, Imagezor uses Q16.16 fixed-point arithmetic, which is shown to give virtually the same network accuracy as floating-point arithmetic[**?**] [**?**] [**?**].

In the sections below we will provide a more detailed description of the convoluter, the hyperblock tangent unit and the max pooler.

4.1.1 The Convoluter

This module is inspired by [**?**]. The input is a $n \times n$ image, and the output is a $(n-k+1) \times (n-k+1)$ feature map, using a $k \times k$ kernel. The kernel is stored in internal registers that must be rewritten for each different feature map that is to be computed. Every clock cycle the module takes in a pixel as input, and after a certain delay it will output a processed pixel almost every cycle. Each pixel is inputted once, left to right, one row at a time.

It consists of 2D grid of multiply and accumulate (MAC) units which represents the convolution kernel. Thus the grid dimension is equal to the kernel dimension. In every MAC unit there is a register that contains the respective

kernel weight. In every clock cycle the MAC units multiply the input pixel with its weight, and then accumulates the result from the previous cycle of the MAC unit to the left.

At the end of each row of MACs there is $n - k$ shift registers. The result of the last MAC in each row is stored in the first shift register, and the first MAC in each row takes the value of the last shift register of the previous row as accumulation input. The exception being the absolute first and last MAC unit. Every clock cycle the values in the shift registers are shifted to the right.

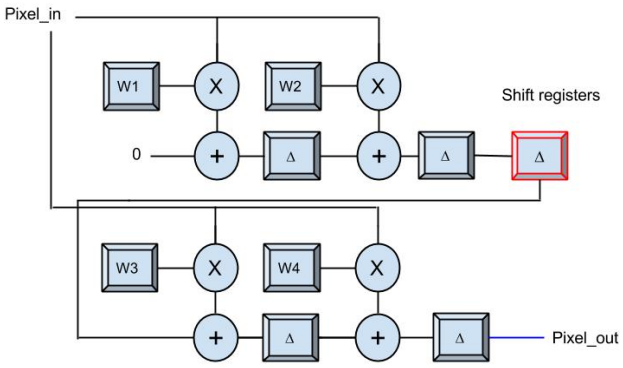

Figure 4.2: The convoluter, when $n = 3$ and $k = 2$.

By providing this delay you only have to input each pixel once during the convolution. Generally every pixel is needed for $k \times k$ convolution operations (the exception being the pixels close to the boarders of the image). Thus the shift registers are used to store the intermediate values of the convolutions until a pixel that is needed for the respective convolution operation is inputted.

The delay these shift registers cause are the reason for the delay before valid output pixels are produced. Thus from when the convolution starts, the output will not be valid before $k - 1$ rows of the image have been processed. And for every new image row, there will be a $k-1$ cycle delay before the output is valid. This is demonstrated by the fact that the input image is a $n \times n$ matrix, while the output matrix is a $(n - k + 1) \times (n - k + 1)$ matrix.

Since the two layers in the network have different image sizes, but uses the same kernel size, we can reuse the module for both of them. This is done by having the control signal *layer_nr* decide how many of the shift registers that are to be used during convolution. In the first layer all of the shift registers are used, but in the second only a subset is used. I.e. $n - k + 1$ of shift registers are used in each row, where n is either 32 or 14.

The loading of the weights takes $k \times k$ clock cycles, and the processing of the image takes $n \times n$ clock cycles. Thus the total number of cycles it takes to perform a full convolution of an image is $n \times n + k \times k$. But based upon the papers refered to in Section **??** it seems that n tends to be larger than k. E.g. for the first layer in the LeNet-5 [**?**], $n = 32$ and $k = 5$, the loading of the

weights take 25 clock cycles and the image processing 1024 cycles. This means that the execution time of the convoluter is primarily bounded by the size of the image. But the size of the kernel decides the hardware resource cost of the module, since it requires $k \times k$ DSP slices on the FPGA. Thus the 32 DSPs on the Spartan 6 is enough in order to implement the LeNet-5.

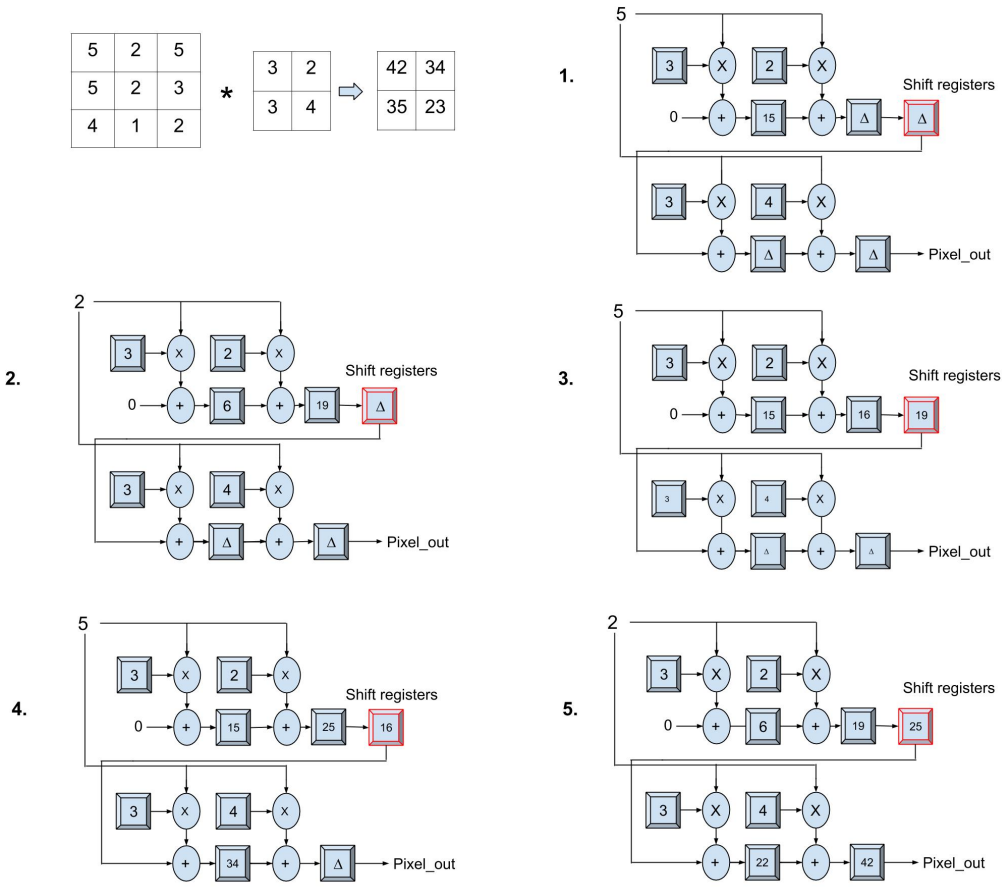

Figure 4.3: Example showing the five first clock cycle of an convolution. The weights of the kernel is already loaded into the MAC units, and every cycle a new pixel from the image in inputted. In the last example you can see that 42 is provided as the first output.

4.1.2 The Hyperbolic Tangent

This module is based upon [**?**] using *piecewise linear approximation*. It takes input a single value x and outputs a linear approximation of the hyperbolic

Conditions	Output
$0 \leq \|x\| \leq a$	$sign(x) \times [0.5 \times m_1 \times \|x\|^2 + c_1\ times \|x\| + d_1]$
$a \leq \|x\| \leq b$	$sign(x) \times [0.5 \times m_2 \times \|x\|^2 + c_2\ times \|x\| + d_2]$
$otherswise$	$signed(x)$

Table 4.1: The piecewise linear approximation of the hyperbolic tangent.

Constants
$m_1 = -0.54324$
$m_2 = -0.16957$
$c_1 = 1$
$c_2 = 0.42654$
$d_1 = 0.016$
$d_2 = 0.4519$
$a = 1.52$
$b = 2.57$

Table 4.2: The piecewise linear approximation of the hyperbolic tangent.

function. Using a lookup table (Table 4.2) the module decides which linear approximation to use.

4.1.3 The Average Pooler

The max pooler performs the subsample/max-pooling operation described in Section 2.2.1. The input is a $(n-k+1) \times (n-k+1)$ feature map, and the output is a $(n-k+1)/p \times (n-k+1)/p$ subsampled/max-pooled feature map, where p is the dimension of subsample neighborhood. As with the convoluter, one pixel is streamed in every cycle, and streamed out whenever a valid pixel is ready. Designed this way the max pooler can process in parallel with the convoluter, by directly streaming the output of the convoluter into the max pooler module. In other words, the operations of convolution layer and subsampling/pooling layer is pipelined.

The module compares the input with the current max value, and updates the max value accordingly. It contains a set of *(n-k+1)/p* shift registers. Since the image is divided into $p \times p$ non-overlapping neighborhoods, the module needs to store the current maximum value of previous neighborhood when a pixel from a new neighborhood is inputted. To do this the module contains two counters, *row_num* and *column_num*. When a new pixel is inputted the *column_num* counter is incremented, and when a new row is encountered the *row_num* counter is incremented. Every time $column\_num\ mod\ p = 0$ the shift registers are shifted one to the right, and every time $column\_num\ mod\ p = 0$ and $row\_num\ mod\ p = 0$ a valid output is produced.

The execution speed of the max pooler module is bounded by the size of the

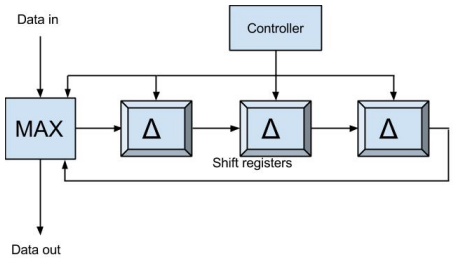

Figure 4.4: The max pooler.

feature map, $(n-k+1) \times (n-k+1)$ clock cycles, finishing one cycle after the last pixel has been inputted. Thus by streaming the output of the convoluter to the max pooler, both will finish only a few cycles apart, effectively running both jobs in parallel. The resource usage of the module is bounded by the size of the subsampling dimension, since it requires a number of shift registers equal to the size of the dimension. But essentially its resource usage is quite low.

Chapter 5

Results and Discussion

5.1 Results

Chapter 6

Future work

Chapter 7

Conclusion