FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Accelerating the training of a Convolutional Neural Network

Afonso Reis

WORKING VERSION

Integrated Masters in Electrical and Computer Engineering

Supervisor: João Canas Ferreira

August 3, 2019

Resumo

Esta dissertação tem como objetivo a implementação de uma Rede Neuronal Convolucional, numa plataforma **FPGA**.

Atualmente, as redes neuronais são o tipo de algoritmo mais usado em aprendizagem computacional profunda. Este tipo de rede foi originalmente concebido para lidar com problemas que involvam imagens, adicionando técnicas de processamento de imagem ao algoritmo original.

O sistema escolhido foi a maxeler, pois este possibilita a escrita de código de alto nivel, libertando tempo para testes e simulações e também porque só existe uma outra implementação com **arquitetura configurável** que seja **capaz de treino** usando este sistema. A solução proposta resolve alguns problemas existentes, nomeadamente em termos de configuração de rede, parallelismo e recursos do equipamento apresentando ao mesmo um *speed-up* de $\times 1.4$. Adicionalmente, uma arquitetura de treino que usa o todo o sistema é proposta.

Abstract

The objective of this dissertation is to implement a Convolutional Neural Network in an FPGA platform.

Neural Networks are, at the time of writing, the most used algorithm in machine learning and deep learning. This particular type of network was originally created to deal with problems regarding images and includes some techniques from image processing.

The maxeler system was chosen because it allows for higher level code to be written which in turn means more time is free for system simulations and also because there's only been one other implementation capable of training using this system. The implemented solution extends on the only other realization, which had some limitations , provides a $\times 1.4$ speedup over the previous design and proposes training dataflow to use available DFE's and the CPU in parallel. Furthermore, the design is fully customizable including network architecture, parallelism and hardware resource usage, via a created API.

Agradecimentos

Agradeço aos meus pais irmão e irmã, por sempre me terem suportado em tudo e me terem dado força sempre que precisei.

Agradeço ao Orfeão Universitário do Porto pelas inúmeras amizades, experiências e lições de vida que me proporcionou.

Agradeço a todos os professores que se cruzaram comigo no meu percurso académico, especialmente ao João Canas Ferreia, que proporcionou tudo e sempre me ajudou da melhor maneira.

We acknowledge the support from the Maxeler University Program regarding the donation of a Galava board and of MaxCompiler licenses.

Afonso Reis

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."

Alan Turing

Contents

List of Figures

List of Tables

Abbreviations and Symbols

CNN	Convolutional Neural Network
ML	Machine Learning
DL	Deep Learning
FPGA	Field Pogrammable Gate Array
GPU	Graphics Processing Unit
ANN	Artificial Neural Network
Tanh	Hyperbolic Tangent
ReLu	Rectified Linear Unit
GD	Gradient Descent
SGD	Stochastic Gradient Descent
MSE	Mean Squared Error
CE	Cross Entropy
DFE	Dataflow Engine
CPU	Central Processing Unit
FMem	Fast Memory
LMem	Large Memory
I/O	Input/Output
API	Application Programming Interface

Chapter 1

Introduction

In this document, a hardware implementation of a **Convolutional Neural Network** (CNN) is presented, with fully customizable architecture and hardware resource usage, using the maxeler dataflow system.

1.1 Background

Neural Networks are, at the time of writing, the most popular algorithm in **Machine Learning** (ML) and were inspired by the main element of the human brain - **neurons**. They are capable of human level performance in a variety of problems. Despite this, they have a few shortcomings, particularly when it comes to training, due to the amount of parameters involved when using large networks, and architecture design, because there is no way to know *exactly* what configuration is best to tackle a problem.

This issue can be mitigated by reducing the desired range of applications and only considering problems that deal with images. By applying some techniques from image processing, such as convolutions, which use a reduced number of parameters, and pooling, to reduce spatial dimensions, in 1999, one of the first CNN's was created [1]. A deeper architecture was proposed in 2012 [2], which won the *ImageNet* competition, proving the relevance of not only this type of neural network, but also of **Deep Learning** (DL). Ever since then, various different types of CNN's have been state of the art methods for any type of problem dealing with images, as attested in chapter 3.

1.2 Motivation

At current time and as far as one can tell from literature, there's only one implementation of CNN's with *customizable architecture* using maxeler's dataflow system [3], and there are no **Field Programmable Gate Array** (FPGA) based implementations which are **focused on training** and have not only a **fully customizable architecture**, but also **fully customizable resource usage**. Therefore, it is necessary to study and evaluate how good the maxeler technology is compared

to other realizations. Software implementations are limited by how many threads can be ran on each CPU core. Because of the parallelizable nature of most required operations, hardware solutions can increase performance, but are limited by chip size, resulting in either small networks or external memory usage. Furthermore, mapping to a **Graphics Processing Unit** (GPU) or to an FPGA requires coding in specific languages, CUDA/OpenCL and HDL respectively, both very low level languages. Therefore, frameworks that allow for configuration and automatic mapping on demand are very useful and are one the leading points in current research.

1.3 Objectives

As mentioned in section 1.2, the main objective is **training acceleration**. Secondary objectives are **network customization**, **resource customization** and **energy efficiency**. **Area efficiency** is not as much of a concern, since the design is not meant for embedded systems.

1.4 Document Overview

In chapter 2, the groundwork is explained in detail, starting with Machine Learning in section 2.1, followed by **Artificial Neural Networks** (ANN) and CNN's.

In chapter 3, previous research is examined, as a means to infer some design choices to ease the implementation.

In chapter 4, the strengths and weaknesses of the maxeler system are described (section 4.1). Section 4.2 describes the entire hardware implementation, including details of each constituent module (section 4.2.7) and memory management in section 4.2.8.

Chapter 5 presents the results achieved by using the created system.

Finally, chapter 6 presents an overview of the entire architecture, as well as some future work suggestions.

Chapter 2

Theoretical Background

2.1 Machine Learning

Machine Learning is a field of **Computer Science** and subfield of **Artificial Intelligence** focused on exploring and studying algorithms which allow computers to learn procedures from data. The main objective is **feature extraction from sample data**. These features are then used for the creation of an adaptive mathematical model, constantly improving with time, later used to make predictions on real data.

Considering a practical example, such as a system that correctly predicts the name of an animal in a given image. One way a system like this could be built is by creating a set of rules it would check for in every image - paws, wings, tail, among others. This approach is not scalable, because as the number of animals the system is expected to detect increases, so do the features and rules. ML approaches the problem in a different way. A **dataset**, of arbitrary size N (the larger the better) $x = [x_1, x_2, \ldots, x_n]$, is created, from several images of all the animals the model is expected to detect. When creating the dataset, it's important to have the name of the animal present in each image, which are called **labels** $y = [y_1, y_2, \ldots, y_n]$. Upon completion of this step, depending on the problem, it can be useful to apply some preprocessing on the data to extract features from it. In this particular problem, some examples could be edge detection or histogram equalization. Now, the system is ready to be created and trained.

The only remaining question is how does the system learn or, in other words, how does the model evaluate its performance and make improvements. The most usual answer to this question is a **cost function**. This function measures the cost of each classification done by the model. A high cost means the prediction was way off, while a low cost means the prediction was close. The objective of training is to minimize the cost function, so that the model's predictions are as accurate as possible. This problem is what's known as a **classification problem**, which is a subset of the problems ML tries to address, because, for each input , the model tries to fit it into the most appropriate class (discrete output). One can also address **regression problems**, where the output is instead a continuous interval, such as the prediction of the price of a house.

Summarizing, the problems ML tackles are characterized by having a large amount of data

which is used to train a model so that the quality metric, the **cost function**, is minimized. There are several algorithms, each with certain strengths and weaknesses, used in *ML* and this work focuses on one of the most important algorithms - ANN's (section 2.2), as a contextual introduction to the main theme, CNN's (section 2.3). More specifically, the latter is an improvement over the original algorithm, which is especially tailored for dealing with problems involving images. This type of network is very used in DL problems, where the extracted features are more abstract, requiring more complex models than average ML problems.

2.2 Artificial Neural Networks

2.2.1 Introduction

ANN's are systems inspired by the human brain that try to reproduce its way of operation. Their main building block, *the perceptron*, is a high-level model of the biological neuron. Every neuron in the brain is connected by dendrites, which of course means that an ANN also has several perceptrons connected by synapses, which make for a more accurate system.

2.2.2 Forward Propagation

Mathematically, the perceptron takes in the sum of weighted inputs and outputs this sum subject to an activation function.

$$y = f(x \cdot w^T + b) \tag{2.1}$$

In the above equation $x = [x_1, x_2, \ldots, x_n]$ is the input vector to the node, $w = [w_1, w_2, \ldots, w_n]$ is the input weight vector, b is the bias factor and $f(.)$ is the chosen activation function.

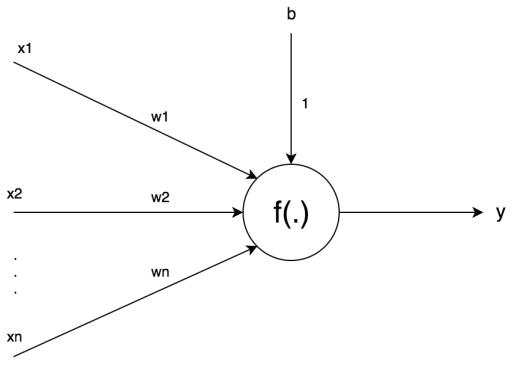


Figure 2.1: Neuron model

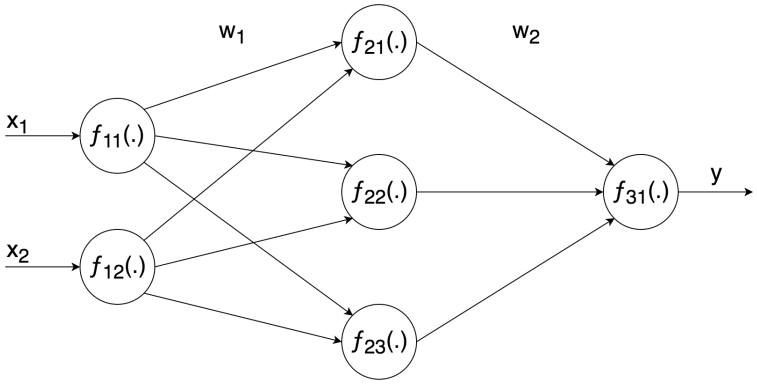


Figure 2.2: 3 Layer ANN

1 - Activation Function

There are several possible choices for activation functions. The first option is the **step function**, which outputs 0 if the input is below a certain threshold and 1 otherwise. This function's output is binary, but there can also be activation functions with real value outputs, such as the **sigmoid** or the **Hyperbolic Tangent**(Tanh), used in [1]. The fact that these functions are differentiable is very important, because the training phase involves derivatives of these functions. As seen in figure 2.3, these functions have a very small gradient outside of the center area, which gives rise to the **vanishing gradient** problem (further described below in the discussion regarding Weights). As such, a new activation function - **Rectified Linear Unit** (ReLu) was created. It outputs 0 if the input is negative and x if the input is positive. This fixes problem with the gradient being low for large inputs, but not for negative inputs. As a further improvement, a small modification was made, regarding the output when x is negative which is now 0.01x instead of 0. This allows the gradient to never vanish and allows for training to continue, which in turn improves convergence. Furthermore, in recent research, instead of $0.01x$, kx is now applied when the input is negative, where k is a parameter learned by the network along with rest.

Table 2.1: Activation functions

Function	Definition	Derivative
Step	$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$	$f'(x) = 0$
ReLu	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$	$f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$
Leaky ReLu	$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{otherwise} \end{cases}$	$f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0.01, & \text{otherwise} \end{cases}$
Sigmoid	$f'(x) = \frac{1}{1+e^{-x}}$	$f(x) = \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}})$
Tanh	$f(x) = tanh(x)$	$f'(x) = 1 - tanh(x)^2$

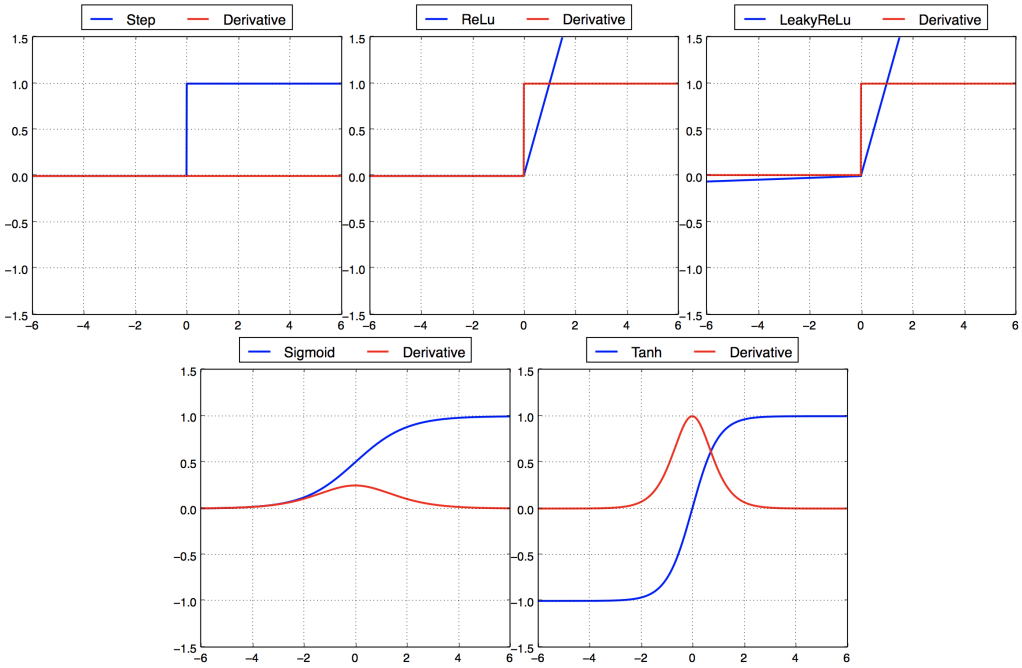


Figure 2.3: Activation Functions

2 - Weights

When initializing a model, the weights are set randomly. However, there are several ways of doing this, the most common way being just taking random numbers from a normal distribution, which can lead to 2 different potential issues - **vanishing gradient** and **exploding gradient**.

Vanishing gradient happens when the input to a neuron is such that the gradient of the activation function is very small, which in turn causes the weight update to be very close to zero, resulting in very slow convergence. On the other hand, **exploding gradient** happens when the gradient of the activation function is large, which causes the weight update to be larger than it should be, resulting in oscillation around the local minimum of the cost function.

[4] proposes another way of randomly initializing the weights. Instead of random numbers, a heuristic is used, depending on the activation function, which eliminates both of the problems described above. The numbers are drawn from a normal distribution with variance $\frac{k}{size_l}$, where k is a parameter depending on the activation function, and $size_l$ is the size of layer l. This is defined as follows:

$$W_i = random * \sqrt{\frac{k}{size_{l-1}}} \tag{2.2}$$

For **Tanh** and **Sigmoid**, $k = 1$.

For **ReLu** and its variants, $k = 2$.

2.2.3 Training

The main objective of training, as said previously, is to minimize the **cost function**, or in other words, to change the network parameters so that the error is minimized. Mathematically, this is achieved by calculating the gradient, equaling it to 0 and solving for the parameters. This is a viable approach only for very small networks. For networks with any sort of practical use, this method is unfeasible as there are too many parameters. As such, a different approach is used, named **Gradient Descent** (GD). The local minimum is reached by moving slowly, in the direction of the negative gradient until the cost is either low enough or not changing. Equation 2.3 is the main building block of GD.

$$w = w - \eta \nabla E(w) \tag{2.3}$$

Training can be divided in 2 major steps - a **feed forward** process where the network is ran, with an input from the used dataset, resulting in a prediction and a **feed backward** process, where, in each layer, from output to input, the error of each neuron is calculated (δ on equation 2.8) and the update for each weight present in the layer. Afterwards, δ is used as the error for the previous layer and the process repeats itself, updating every parameter in every layer. The feed forward step is self-explanatory, and follows equation 2.1 for all the neurons present in the network. During this step, its important that each neuron stores its input, because this value is used during calculations for backpropagation. The output of the network, $\hat{y}$, and a **cost function** to calculate the error at the output layer are also needed. The two most commonly used functions are **Mean Squared Error**(MSE) and **Cross Entropy**(CE).

$$(MSE) \qquad \text{Error} = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}-y)^2 \tag{2.4}$$

$$(CE) \qquad \text{Error} = \frac{1}{N}\sum_{n=1}^{N} y\log(\hat{y}) \tag{2.5}$$

The backpropagation step, however, is not quite as simple. Because of this, to review what's been said so far and to make notation a bit less confusing, below is a list of all the notation used.

1. δ_{li} - the delta value of neuron i at layer l
2. $\frac{dE}{dy}$ - derivative of the cost function.
3. $\frac{dAct_{li}}{dy}(y)$ - derivative of the activation function of neuron i at layer l, evaluated at y.
4. w_{lij} - weight at layer l connecting neuron i of the current layer to neuron j of layer $l+1$
5. η - learning rate
6. y_{li} - output of neuron i at layer l.

The first calculation in a generic network composed by N layers is the delta of the output layer, given by

$$\delta_{Ni} = \frac{dE}{dy}\frac{dAct_{Ni}}{dy}(y_{Ni}) \tag{2.6}$$

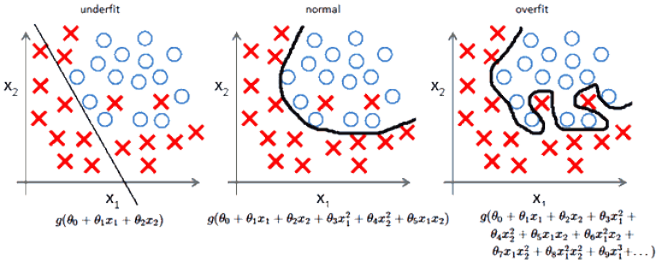
After this and for every layer except the output layer:

$$w_{lij} = w_{lij} - \eta(y_{li} \cdot \delta_{l+1,j}) \tag{2.7}$$

$$\delta_{li} = \frac{dAct_{li}}{dy}(y_{li}) \sum_j \delta_{l+1,j} \cdot w_{lij} \tag{2.8}$$

Mathematically, GD, as described above, would be the most correct approach and would yield the best results. In practice, however, this is not a good method, because, when the dataset is large, too much computation is required for each update, which would make training really slow. This is because of the fact that the entire procedure would have to be repeated for the entire dataset and only then could the updates be applied to each parameter (averaged, of course). [5] presents a solution to this problem, **Stochastic Gradient Descent** (SGD). Before each training iteration, a mini-batch $x_{train} = [x_1, x_2, \ldots, x_k]$ consisting of K is selected from the dataset. This mini-batch has significantly reduced size and is used to train the network.

Both of these methods work well and, if given enough time, converge to a local minimum of the cost function. However, there are two main problems with these methods - **slow convergence** and **overfitting**. Slow convergence has been improved with other training algorithms which will be described below and still remains as the biggest problem faced when trying to optimize a training method. Overfitting can happen in one of two scenarios - the model is too complex for the given problem (figure 2.4a), which can be resolved by attempting several models with different designs and choosing the one that performs best, or the model is trained for too many iterations (figure 2.4b). In either situation, performance is very good on any input in the training dataset and not nearly as good on real data that the model hasn't yet seen - the model doesn't **generalize**.



(a) Overfitting shown by the models' function

(b) Overfitting shown by comparing errors

Figure 2.4: Both images show why overfitting can be a very serious problem when used in applications where the error needs to be as low as possible.

Source:StackExchange

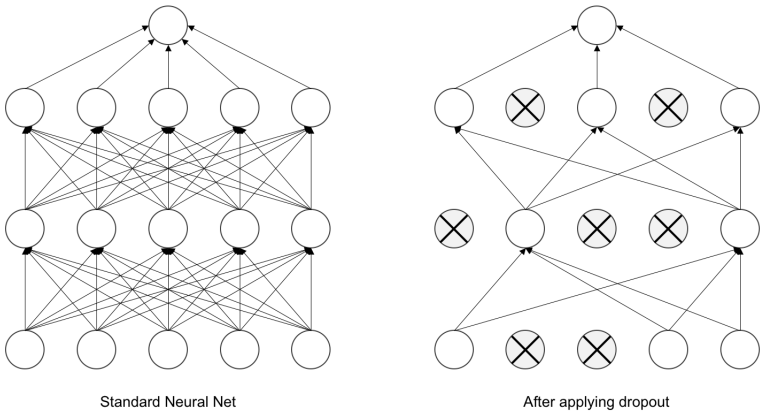


Figure 2.5: Dropout applied to a standard ANN. Usual values for dropout probability are within the 50-75 % range

Source:Deep Learning

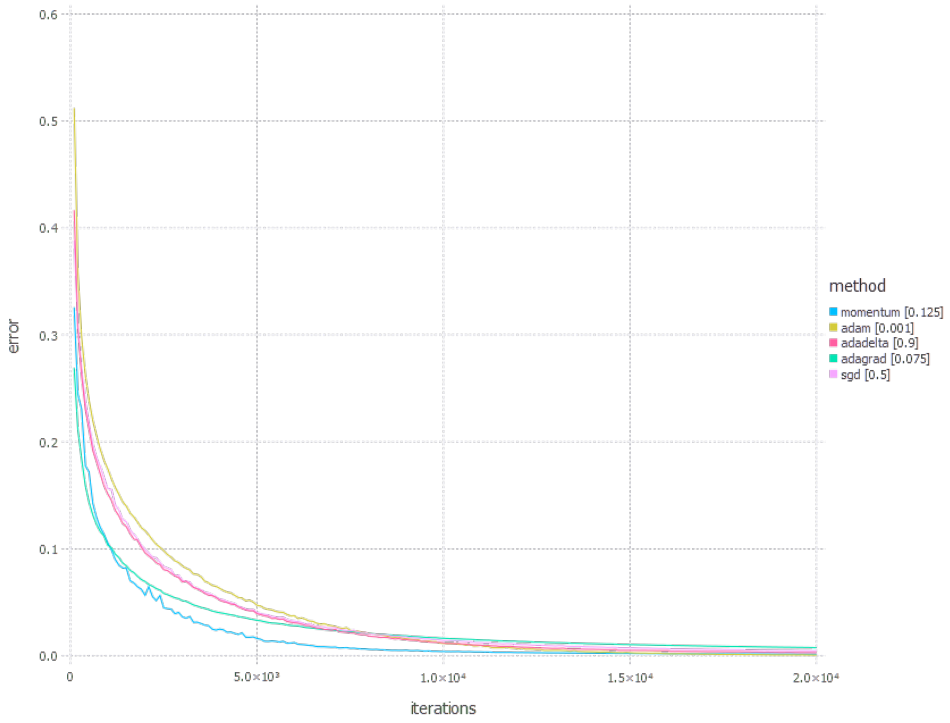
There are a few solutions to this problem, the two most common ones being **dropout** a concept originally described in [2] applied during training, where neurons are randomly "turned off", setting the respective output to 0 regardless of input, with probability p, as shown in figure 2.5, forcing the network to learn different paths to each output and **early stopping**, described in [6], which, as the name implies, means training is stopped before the model gets too used to training data, as demonstrated in figure 2.4b.

It's possible to achieve faster convergence by adding more hyperparameters and changing the training equations slightly, requiring more computation per iteration but reducing overall time spent training until convergence is reached. The first method created using these extra parameters is an extension of SGD, and its name is that of the added parameter, **momentum**. The only change this has is in equation 2.7. $update_i = (y_{li} \cdot \delta_{l+1,j})$ is how w_{lij} was updated at training iteration i. The new update rule for the weights is defined below, where γ represents momentum.

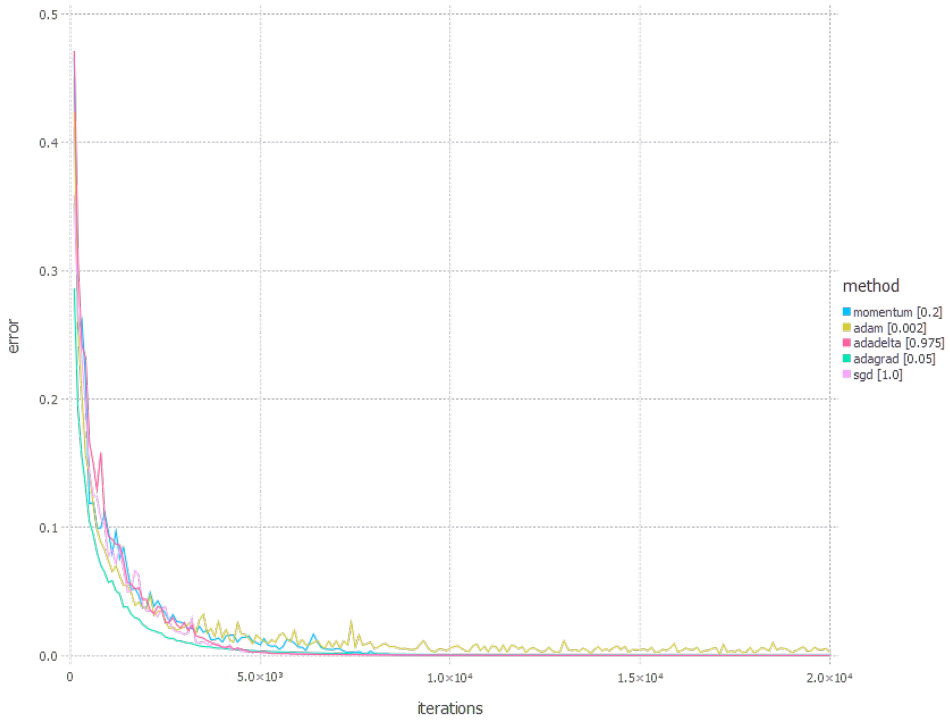
$$w_{lij} = w_{lij} - \eta \cdot update_i - \gamma \cdot update_{i-1} \tag{2.9}$$

This update rule was originally experimented with because of the fact that, if the gradient is pointing in the same direction 2 iterations in a row, faster movement could be done in that same direction. If it points in different directions, a local minimum has been reached and oscillation is happening around said value, so slower movement is required. In both cases it's favorable to use and reduces the overall time required to reach convergence. Despite all this, momentum also has a few drawbacks, the main one being during initial iterations the model might make really big jumps, missing the global minimum and converging into a local minimum instead.

Recently, there's been a lot of research revolving around training algorithm optimization and there have been discoveries which fix these problems and even some that introduce a new concept of altering the **Learning Rate** and **Momentum** dynamically. [7] presents a concise overview of training algorithms, the most commonly used ones beeing **Adam**, **AdaDelta** and **AdaGrad**.



(a) Fcon layer with ReLu, followed by Softmax



(b) Fcon layer with Tanh, Fcon layer with ReLu, followed by Softmax

Figure 2.6: Training Algorithms compared, with two different ANN architectures

Source:Training Algorithm Comparison

2.3 Convolutional Neural Networks

2.3.1 Introduction

A Convolutional Neural Network (CNN) is an extension of a regular ANN especially tailored for dealing with high dimension data. This type of networked was theorized by expanding on a concept called "sparsity", which means each neuron is only connected to a few neurons in the next layer or, in other words, each neuron is only **locally connected**. With this technique, a new layer called the **convolutional layer** was adopted into the model. In addition, to reduce spatial dimensions, **pooling** was also adopted into these networks. Although originally a technique from image processing, it works well in CNN's, to reduce training time. In section 2.3.2, all the layers used to compose a CNN will be be explained in detail.

2.3.2 Layers

1 - Activation Layer

This layer is responsible for the activation function. There are usually many of these in a CNN, and the most common activation functions are the same as described 2.2.2. Although each activation layer can have a different activation function and even each neuron in each layer, due to the increased complexity in network structure, ReLu and its variants gain more popularity and are used the most, not only because of simplicity but also to decrease training time. In addition, another activation function, **SoftMax** is used, only once, on the last layer, to convert the final feature vector into a normalized N dimensional vector, where each position contains the probability of the input corresponding to a certain class.

Softmax is defined as follows:

$$y_i = \frac{e^{x_i}}{\sum_i e^{x_i}} \tag{2.10}$$

with respective derivative

$$\frac{dy_i}{dx_k} = \begin{cases} y_i(1-y_i), & i = k \\ -y_i y_k, & i \neq k \end{cases} \tag{2.11}$$

This result can be combined with any cost function, but is usually combined with CE, since the expression becomes a lot simpler:

$$\frac{dy_i}{dx_k} = y_i - l_i \tag{2.12}$$

[8] elaborates further on each of these definitions, with practical examples, as well as complete derivation of these expressions.

2 - Convolutional Layer

This layer is one of the most important in CNN architectures, as it's responsible for feature extraction. This is achieved through the **convolution operation**. Usually, there are many of these layers in a CNN and they are almost always followed by an activation layer.

The main building block of a convolutional layer are **kernels**, which function as the *weights* of a CNN. Each convolutional layer has many kernels, each with different values. Each one has dimensions $C \times N \times N$, where C represents the amount of channels the input has (3 in the case of an RGB image, for example) and N represents the kernel size which is usually an odd number, so the kernel has a center. Each kernel then goes through the entire input like a "flashlight", multiplies its values by the input values in the highlighted area, sums them and generates an output, as seen in figure 2.7. The output of each kernel is usually called a feature map.

Early layers usually detect lower level features, which are then combined by later layers into higher level features, which are used by fully connected layers to classify the image. Alternate classifiers can also be used instead of the chain of fully connected layers usually present at the end of the network. Some examples are Support Vector Machines or Random Forest.

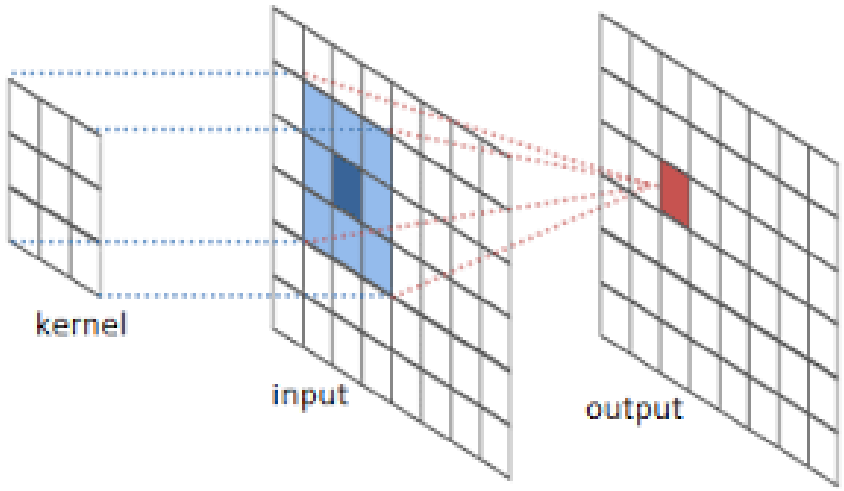


Figure 2.7: Convolution Operation

Source:Convolution

This first consequence of doing convolutions regards forward propagation - the output does not have the same spatial dimensions as the input. More precisely, the number of channels will be equal to the number of kernels associated with the specific layer and the output size goes from $C \times N_{input} \times N_{input}$ to $1 + (N_{input} - N + 2 * Pad)/Stride$, where Pad is the amount of padding introduced to the input and stride is how many pixels to the right and down the kernel moves each time. For example, if $Pad = 2$, as shown in figure 2.8 and $Stride = 2$ the top left corner of the kernel begins at position (x_1, y_1) of the padded input, the next positions will be (x_1, y_3).

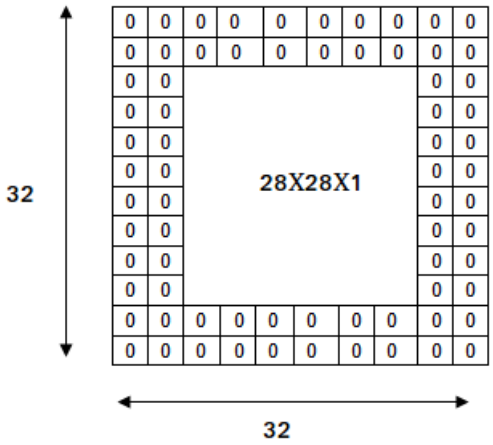


Figure 2.8: Zero Padding operation with Pad = 2

Source: Padding

The second consequence regards backward propagation - the output needs to be scaled up, because it was scaled down during forward propagation. This is achieved by a different kind of convolution, the **Full Convolution**, shown in figure 2.10. This however, is only the case if *stride = 1*. Otherwise, yet another type of convolution is needed, the **Fractionally Strided Convolution**, also known as the **Transposed Convolution**, shown in figure 2.9. Both convolutions are done with the kernels flipped and the δ from the next layer. Both Input and Output sizes follow the same rules as they do in forward propagation.

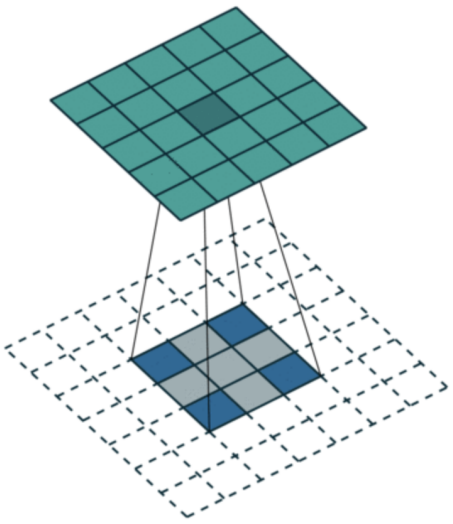


Figure 2.9: Fractionally Strided Convolution. Blue squares are values from Delta. Green squares represent Padded Delta. White squares are zero and are present because Stride = 2.

Source: Fractionally Strided

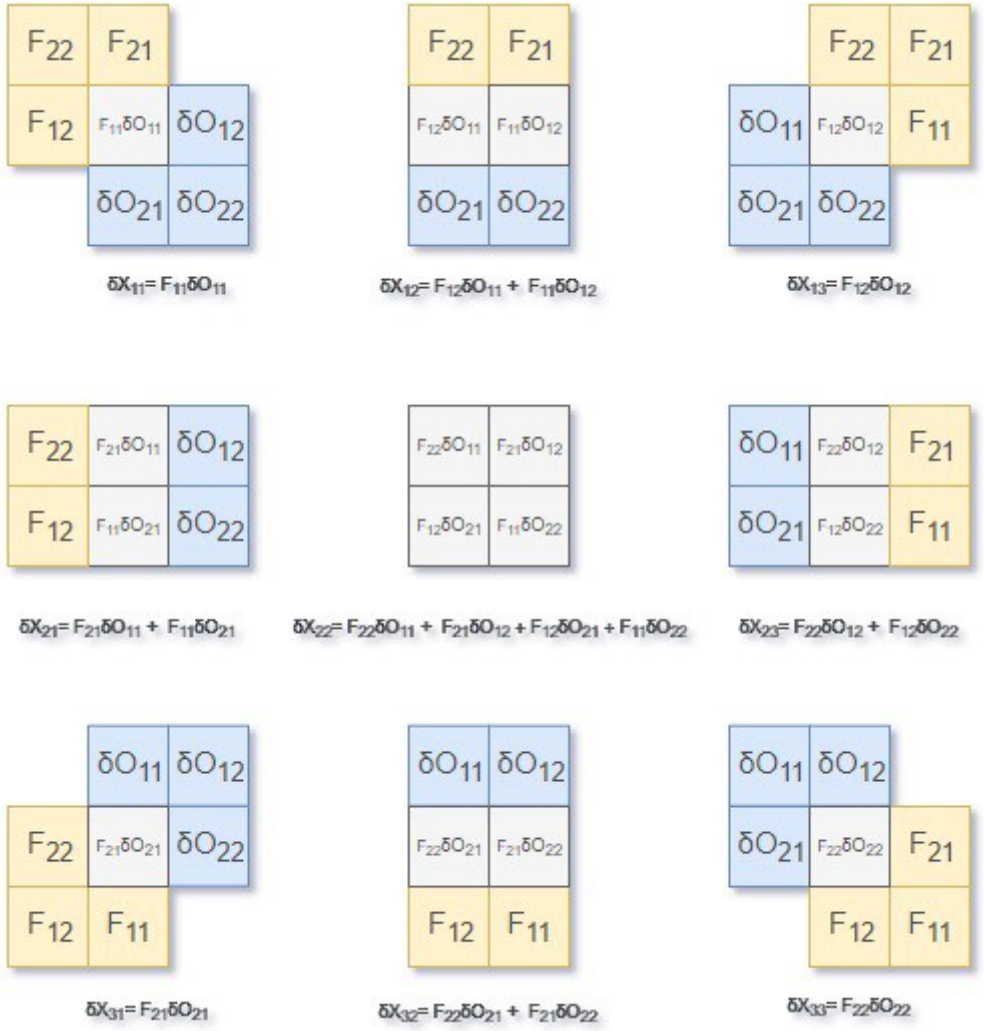


Figure 2.10: Full Convolution. Yellow squares are kernel weights, blue squares are delta values.

Source: Full Convolution

3 - Dropout Layer

This layer works exactly like **dropout**, previously described in section 2.2.3 and is used to prevent the model from overfitting, which can be an even bigger problem in CNN's, because of the increased model complexity. They are generally only used in after fully connected layers, due to the fact that almost all parameters in the network are present in these. p usually has values around 50%.

4 - Fully Connected Layer

Once again, this layer works exactly like a regular ANN, previously described in section 2.2. It receives an input and multiplies it by the weight matrix. Usually, these layers are present after all convolutional and pooling layers and are always followed by an activation layer as their role is not only feature extraction, but also classification with the SoftMax activation function.

5 - Pooling Layer

This layer is responsible for reducing spatial dimensions, thus reducing the amount of needed computational power, and are usually present after a chain of convolutional layers.

Forward propagation is mostly done with one type of pooling - **Max Pooling**, which selects the maximum value in a certain window. There are others, like **Average Pooling** and **L2-Norm Pooling**, but these are not used too often. Figure 2.11 illustrates both Max Pooling and Average Pooling. Similarly to the convolutional layer, this layer also scales the image dimensions down, from $C \times N_{input} \times N_{input}$ to $1 + (N_{input} - \text{WindowSize} + 2 * \text{Pad})/\text{Stride}$.

Backward propagation involves upscaling. If max pooling is used, the position of the maximum needs to be recorded. Usually this is named "mask" and is set with 1 on the positions where the maximum was found and 0 otherwise. Then, when performing backpropagation, this mask is used and the corresponding elements are multiplied by the gradient computed from the next layer, as shown in figure 2.12. The only change when other types of pooling are considered is the configuration of the mask array.

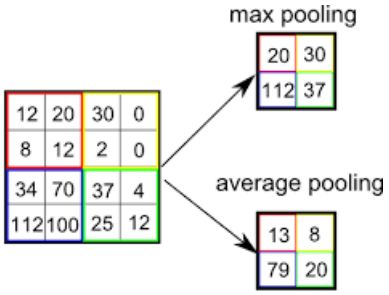


Figure 2.11: Pooling Forward propagation Operation. *Stride* = *2* and *Window Size* = *2*

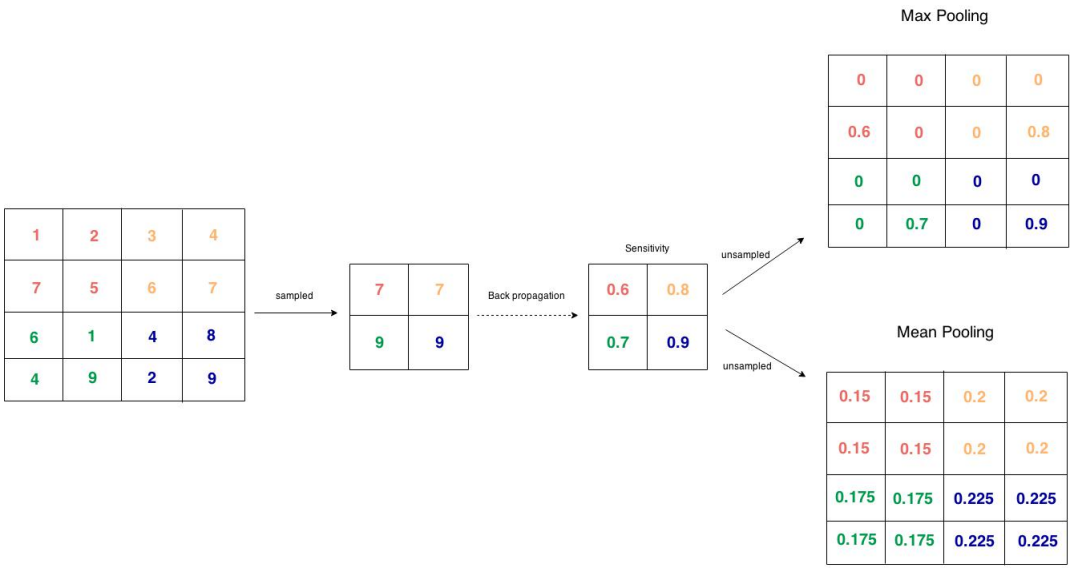


Figure 2.12: Pooling Back propagation Operation. *Stride* = *2* and *Window Size* = *2*

Source:Pooling

6 - Overview

Summarizing, the simplified Mathematical functions for each layer are described in table 2.2.

Table 2.2: Mathematical definitions of each layer

Layer	Forward	Backward
Act	y = f(X)	δ = error $\times \frac{df}{dx}(X)$
Conv	y = conv(Kernels, X)	error = conv(flip(Kernels), X) Kernels -= $\eta \times$ conv(delta, AvgX)

Drop	if(p > random())
	y = 0
	else
	y = X
		Nothing

Fcon	y = X $\times W^T$	error = $\delta \times W^T$ W -= $\eta \times$ X $\times \delta$
Pool	y = pool(X)	delta $\times$= Mask

2.3.3 Architecture

Architecture is the most important part of a CNN. The first layer is always convolutional. From here on out, there are a lot of choices:

The *first choice* is to have an activation layer after each convolutional and fully connected layer. This is almost always the case, because after feature extraction, it's important to separate true features from falsely detected ones. A small optimization can be done when a convolutional layer is followed by a pooling layer, by having the activation layer after the pooling layer, so less calculations are required.

The *second choice* aims to answer the question "How deep should the network be". Largely, this choice is problem dependant, because for small applications even moderately small architectures have good enough performance. [9] and [10] experimented with deeper networks than [2] which resulted in better classification accuracy, but this approach is not scalable, because network complexity increased more than computational power in the same time span. [11] provides a solution with a new module which allows for networks with large amounts of layers (more than 100) to be less computationally complex than previous architectures.

The *final choice* regards fully connected layers. How many should be added and how many neurons each layer should have. At least one has to be present, to be used with SoftMax for classification. By looking at networks that perform well, [2] and [9] show that around 1-4 fully connected layers is a reasonable amount, with neuron counts that are powers of 2. This is because networks are usually trained on GPU's or FPGA's and having powers of 2 as the number of neurons helps increase parallelism.

Figure 2.13: The Architecture of the VGG network. The Softmax layer with 1000 nodes at the output is omitted at the end, for simplicity.

Source:VGG-Arch

Figure 2.13 shows the architecture of the VGG-16 CNN [9]. The input is a $3 \times 224 \times 224$ image, and the output is a 1×1000 vector of probabilities corresponding to the probability of the input image being of that class. There are a lot of interesting design choices present.

The first one is that all kernels are 3×3 in size and increase in amount the later layers. Another curious fact is that the output size of convolutional layers stays the same as the input, which means padding is beeing used to preserve dimensions. A lot of numbers are powers of 2, such as all Kernel amounts and amount of neurons in fully connected layers.

This network was originally created to discover how to best design network structure and what values should be used for the different layers. The results are obvious - the smallest kernel size, 3, is the one used for all convolutional layers, as well as window size 2 with stride 2 beeing used on all the pooling layers. The network has 21 layers total, assuming activation and dropout layers are built into the 3 types which result in roughly around 151 million parameters(136 million in the last 3 layers), making it very computationally expensive to train. This design choice, making the network deep and using simple convolutions and pooling was one adopted by future designs.

2.3.4 Training

Training a CNN is the same as training an ANN, which was described in section 2.2.3, with the exception of the new layers, convolutional and pooling, whose backpropagation process was specified in section 2.3.2.

Chapter 3

State of the Art

In this chapter, implementations of neural networks will be discussed, with specific emphasis on ones that use the maxeler system. Several choices will be made, such as network architectures used for design validation and testing and the training algorithm of choice. Furthermore, some weak points from previous designs will be discussed, as well as possible improvements.

3.1 Convolutional Neural Networks

3.1.1 Network architecture

One of the main applications for a CNN is image recognition, classification and segmentation. ImageNet holds one of the largest image classification competitions each year and, since 2012, every winning model has been a CNN variant.

[2] is first instance of a CNN winning the competition. The network architecture is reasonably small when compared to today's standards. The results achieved were far better than any other previously state of the art models, with a top-5 error rate of 15.3%, compared to 26.2% achieved by the second best entry. The entire training took around 1 week using 2 GPU's. The used training algorithm was SGD, with momentum set to 0.9 and learning rate 0.01, which was lowered three times during the entire training process.

Both [9] and [10] expand on the previous design, the first focusing mainly on exploring deeper network architectures with simple parameters and smaller filter sizes, such as 3×3 filters for all convolutional layers and 2×2 windows with stride 2 for all pooling layers, while the second focuses on optimizing computing resources, by implementing local connections between neurons of each layer. Both are trained using SGD with momentum.

Both implementations are combined in [11], which uses both simple network parameters and sparse connections between neurons of each layer. This results in very deep networks, of up to 200 layers, with less complexity than [2], resulting in faster training and better accuracy. The chosen training algorithm is, once again, SGD with momentum.

3.1.2 Maxeler Implementations

[3] is the first CNN implemented on a maxeler system. There are several strong points in this design, such as the customizable network architecture, energy efficiency, performance and data transfer scheme. Although the variable architecture is a strength which should be adopted into other implementations focused on providing frameworks for CNN creation, this is also the main drawback, because the final design uses a layer by layer architecture, introducing a lot of unnecessary overhead when computing. The data transfer scheme works well, due to the fact that the design bottleneck is not computation, but data transfer bandwidth between off-chip memory and the FPGA.

There are still a few unanswered questions, mostly regarding hardware configuration. While [12] only considers inference, the solution provided has variable bitwidth, as well as some insight on computation to communication ratios, to help maximize performance. [13] presents a hardware implementation of a long short-term memory neural network, with a way to customize resource usage at compile time. This is done by computing in parallel according to a user parameter, which is parameterizable.

3.2 Design Implementation

Guven all of the available information and considering the design goals, there are a few design choices that can be made beforehand.

The main bottleneck of the design in [3] is the layer by layer implementation. One possible workaround is to have more than one layer in each module, or in other words, making each module contain a block of layers instead of a single layer. The presented system dataflow scheme is one of the design strengths and will be used, with slight modifications to account for the block architecture. The easiest way to allow for user customization is to create a framework which allows the user to specify network architecture, bitwidth and parallelism as design parameters which will then be used to create blocks to be mapped onto hardware. These parameters will then be accessed by the compiler and be used as ways to completely customize the system, depending on user requirements and available hardware.

Another takeaway from [3] is that the weights are updated on the **Central Processing Unit** (CPU), while the next module is beeing loaded. This means that nonblock functions are available and that the CPU can be freed while the FPGA is computing. Depending on the available CPU, it can be worth it to also perform forward and backward propagations entirely on the CPU, in parallel with the FPGA. As such, implementing a software version of a CNN may increase overall performance and will also be explored when testing the entire system.

The chosen training algorithm is SGD with momentum, due to the fact that [2], [9], [10] and [11] all use it with great results. Finally, regarding architectures used for design validation, [2] and [9] have the simplest architectures and were chosen, because [10] and [11] use more complex layer connections which would complicate the design.

3.3 Overview

As one could attest from this small literature survey, although there are CNN implementations using the maxeler system, there is a great deal of room for improvement, mostly regarding flexibility. Some key design choices were also made, based on strengths and weaknesses from previous designs.

Chapter 4

Proposed Architecture

Over the course of this chapter, a small introduction to the Maxeler system is presented in section 4.1, with respective strengths and weaknesses. Subsequently, in further sections, the CNN design will be explained in detail including the full system overview and design implementation.

4.1 Maxeler

The maxeler system, whose architecture is illustrated in figure 4.1, was created to accelerate applications, by using atleast one **Dataflow Engine** (DFE), which consists of an FPGA with some additional external memory named **Large Memory** (LMem), and the concept of **Multiscale Dataflow Computing**. In these applications, each program is considered as a dataflow graph of executable actions, which are performed as soon as all required inputs are ready and outputs forwarded to the next action in the graph. This process is repeated until every action has been executed. Furthermore, the CPU also plays an important role in the system architecture.

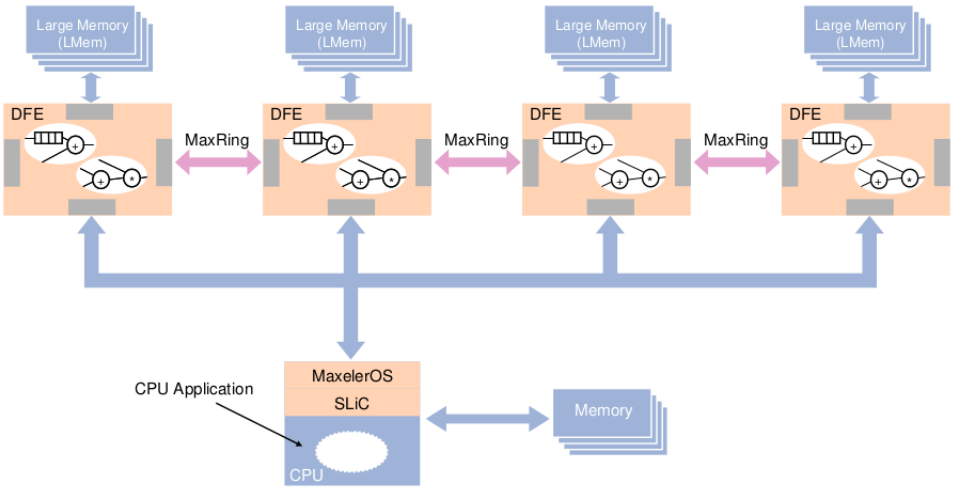


Figure 4.1: The Maxeler System

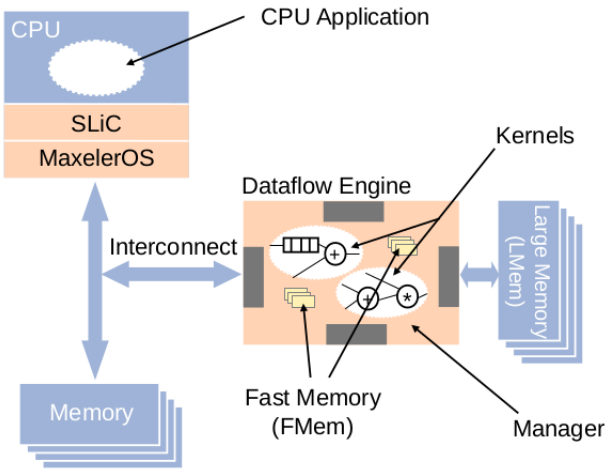


Figure 4.2: Maxeler system used for this thesis

Source: Maxeler

Figure 4.2 shows the system used during this work. It's composed by a Desktop running CentOS 7 and one DFE, more specifically the **Galava** model.

4.1.1 Hardware Resources

Before going into details about the CNN implementation, it's valuable to know the available resources in the Galava card, as a means to understand some of the design choices made. There are 3 ways to move data to the DFE:

1. **FMem** - fastest available memory, not only because of the fact that it's implemented on the FPGA, but also because it's single cycle access. However, it's limited to only 5.6 MB. To access, one must call either read or write with an address.

2. **LMem** - slowest available memory, with a peak data transfer rate of 32GB/s. The technology is DDR3 and the main limitation is not size, unlike the FMem, as there are 12 GB available. When configuring the manager, no more than 15 **Input/Output** (I/O) streams to the LMem can be created. In addition, the LMem has a BurstSize (192 Bytes) - any stream created has to either read or write in multiples of the BurstSize. To access, one must call input or output on a certain stream, which will return one value in memory every tick.

3. **Interconnect** - slowest way to send data to the DFE, with a peak transfer bandwith of only 2GB/s. In an efficient design, it's important to minimize the amount of data sent through this connection. When sending data, one can choose to send streams or scalars. When sending streams, they behave the same way as an LMem stream. When sending scalars, the data value will be the same for all ticks.

Regarding computation resources, there are 490000 logic elements and 512 multipliers (signed 18×18 bit) available. In reality however, since MaxelerOS uses a small amount in the manager, there are a few less resources than said.

4.1.2 MaxelerOS

The **Manager** orchestrates data movement within the DFE, such as I/O data streams from/to the CPU or LMem, any type of multiplexing between said streams and connections between these and existing kernels. In addition, it's also responsible for creating any interfaces between the CPU and the DFE which can be used to perform any DFE operation, such as running kernels or accessing the LMem. **Kernels** run computations, using data from the CPU, the LMem and the FMem.

The **CPU** has access to any interfaces created by the manager, which require certain parameters to be ran. As a result of this, in advanced designs, an **Application Programmable Interface** (API) is usually created, which allows the user to customize the design and automatically configures all required parameters for DFE runs and obviously, allows for DFE runs. The interfaces created by the manager can be ran in two modes - **blocking**, which holds the CPU while the DFE is running and **non-blocking**, which instead frees the CPU.

4.1.3 Data Streams

Data on the DFE, with the exception of the FMem, is handled via streams. Considering an input stream of N data points $x = [x_1, x_2, \ldots, x_n]$, reading during tick t returns x_t. As said in section 4.1.1, scalar streams work the exact same way as an input stream, with the exception that every point in the stream has the same value. By the same logic, an output stream of N data points $y = [y_1, y_2, \ldots, y_n]$, writing during tick t places the write value on point y_n.

If multiple points of a data stream are required to perform a specific computation, for example the addition of x_n and x_{n+3}, they can be easily be accessed by using offsets, which can be either positive (stream values from the future) or negative (stream values from the past). In this case, using an offset of 3, which requires the DFE to store 4 points $[x_n, x_{n+1}, x_{n+2}, x_{n+3}]$, solves the problem. This means that, **no matter how many points are required per tick**, the amount of points stored by the DFE is $|MaxOffset| + 1$.

In most applications, some kind of input control and output control is required. This kind of control can be used on streams. Data streams can be stopped, by **reading or writing with an enable signal**. Reading returns the next point in the data stream if enabled, or the same point if not. For example, if enable alternates between 1 and 0, the resulting input stream is $x = [x_1, x_1, x_2, x_2, \ldots, x_n, x_n]$, requiring twice as many ticks to reach the end. This must also be taken into account when using offsets, because the input stream if modified. Considering the previous example, offset would now have to be 6. Writing with an enable signal simply determines ticks where data is written and ticks where data isn't written.

4.2 CNN Design

4.2.1 Resource Usage

In accordance with the defined objectives, one of the main goals is a highly configurable system. As such, there must be a way to use more or less resources, depending on how many are available on the used DFE and also depending on the Network Architecture.

4.2.1.1 Runtime Reconfiguration

The maxeler system allows for the reconfiguration of any DFE's used in the system as many times as required during execution. As said in chapter 3, [3] uses a layer by layer approach, which is extremely limited, in spite of the fact that every layer has access to every available DFE resource to run which makes layer propagation extremely fast, because the DFE has to reconfigure itself for every layer. The reported reconfiguration time is anywhere between 100ms and 1 second. Figure 4.3 shows measured reconfiguration times for the Galava DFE. The amount of time spent reconfiguring the DFE can be very costly, especially when dealing with deep networks. This fact instantly rules out the maxeler system when low latency CNN's are required, unless there's one DFE available per layer. Furthermore, the LMem is reset every time the DFE is reconfigured, requiring the output to be read before a layer is unloaded and rewritten when the next layer is loaded

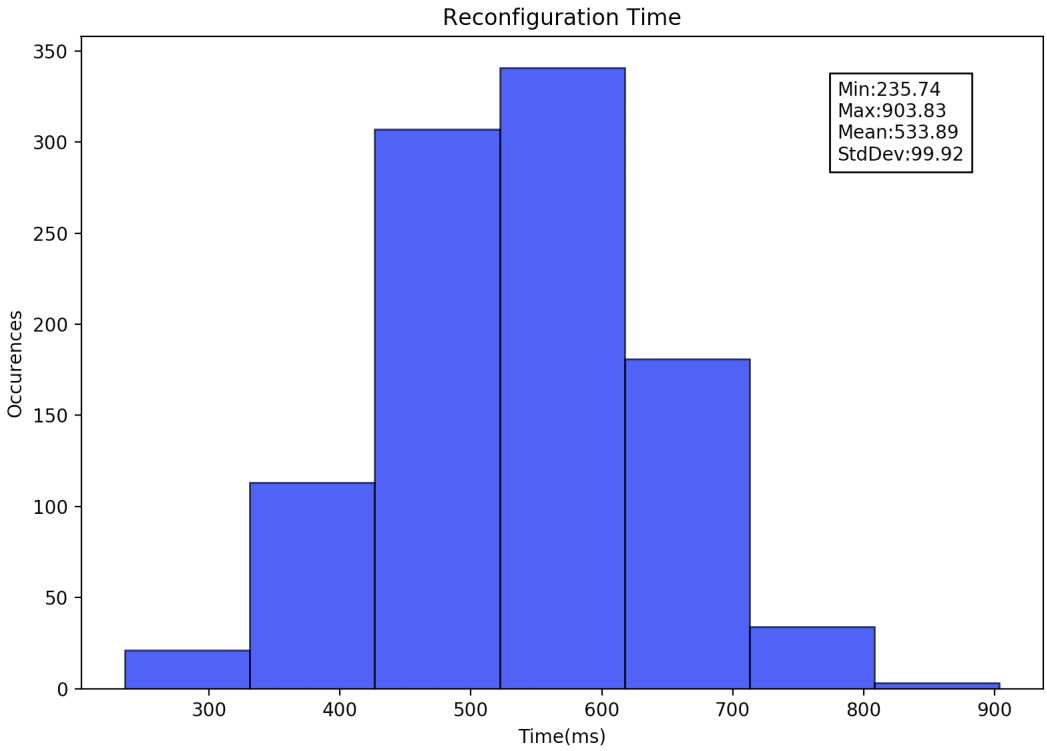


Figure 4.3: Reconfiguration times for Galava

The DFE efficiency of a maxeler system can be described by equation 4.1.

$$\text{DFEEfficiency} = \frac{\text{ExecutionTime}}{\text{ExecutionTime} + \text{ReconfigurationTime}} \tag{4.1}$$

Computation times reported for LeNet5 in [3] are approximately 18.4 ms for single inference. Obviously, running single inference would result in an efficiency, $\frac{T_{Inferece}}{T_{Inferece}+N_{Layers}\times T_{Reconfig}}$, of only 0.57%. Fortunately, in an environment when inference latency isn't important, like training a CNN, where the main objective is speed, there's a simple way to increase efficiency - run more than one inference each time a layer is loaded. This is usually already the case when using SGD, as mini-batch of size B_S is ran each iteration, which means efficiency is $\frac{B_S\times T_{Inferece}}{B_S\times T_{Inferece}+N_{Layers}\times T_{Reconfig}} =$ 68.8% , considering $B_S = 384$, as reported.

4.2.1.2 Network Block Architecture

To improve on the layer by layer implementation of [3], an API was created, in C, which allows for the creation any CNN or ANN architecture in blocks of layers, as shown in figure 4.4. Each block can contain any combination of layers with any valid parameters. During propagation, the output of each block serves as input to the next block.

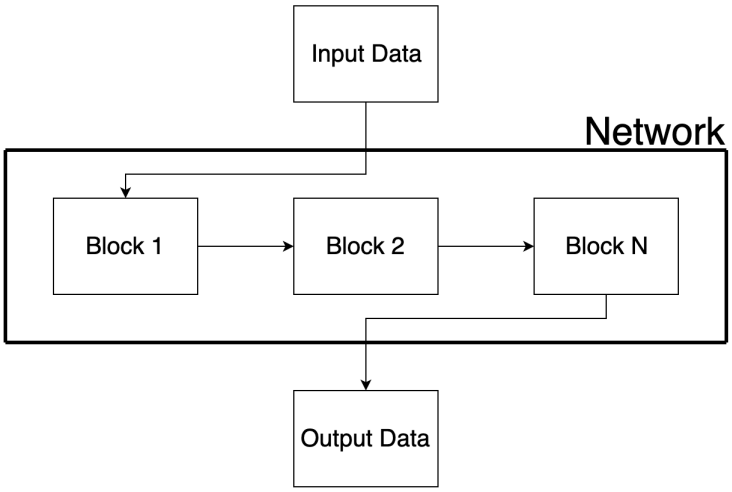


Figure 4.4: Network Block Architecture when performing forward propagation

When using the DFE for computation, blocks are loaded sequentially, one at a time, allowing for both shallow networks where all layers can be in one block and for deep networks using multiple blocks. Furthermore, there are two extra parameters, to increase customization - **BurstMult** and **Parallelism**. BurstMult allows more or less data to be written by the DFE each time the kernel is ran, in multiples of 192, to respect the burst size, meaning the Kernel has to be ran less times for the computation to finish. Parallelism allows convolutional layers and fully connected layers to have access to more or less resources for computation, which in turn makes computation faster or slower.

4.2.2 Data Transfer

In order to minimize CPU to DFE data transfer, a memory scheme similar to [3] was chosen:

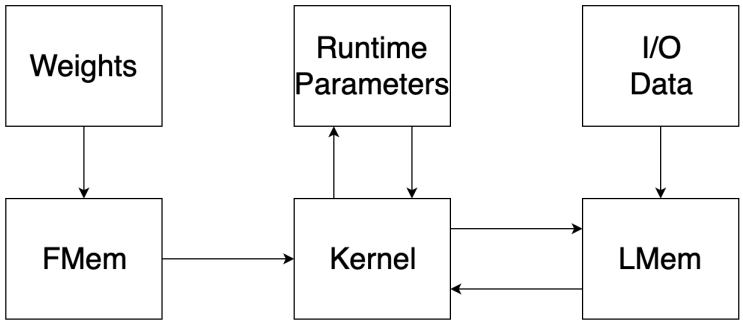


Figure 4.5: CPU/DFE data movement

There are two ways of having large amounts of input data available to the kernel, either the interconnect PCI-e link or the LMem, the latter beeing the logical choice, because of superior speed. Furthermore, the interface used to run the kernel is simplified as the LMem configurations are done with minimal CPU parameters, because of the fact that, if memory streams were an input parameter, the function used to run the kernel would change according to the number of layers in the block. One consequence of using the LMem is that burst size has to be taken into account. [3] uses a clever strategy, specifically for training, by making BS a multiple of 192. This means each layer has input dimensions $BS \times \text{NChannels} \times \text{InDims} \times \text{InDims}$ and also means single inference is very inefficient. There are drawbacks when considering the block architecture, because 12GB of LMem isn't enough to store all the required data on large blocks. Another option is to pad data until the input stream length is a multiple of the BurstSize, resulting in a layout shown in figure 4.6. The 15 I/O stream restriction also has to be taken into account, to allow for at least three layers to be present in each block.

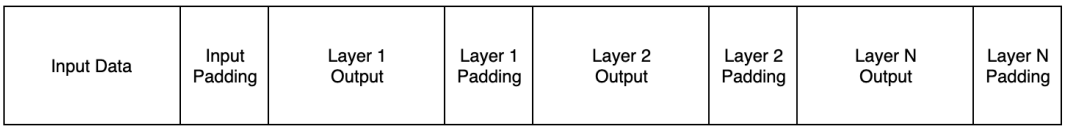


Figure 4.6: LMem layout

Regarding the interconnection link, the amount of parameters has to be minimized. The weights need to be available for the kernel, and the FMem is not large enough to contain all weights. Contrary to expectations, this is not a problem, due to the fact that, even if every weight were available at once, the DFE would not have enough resources to perform that many multiplications. Furthermore, in order for the interface to not vary in definition, the weights are limited to only 1 array, which can, at most, contain 65535 values. When performing backpropagation, weight updates are streamed through the interconnection link, to avoid using an extra LMem stream.

4.2.3 Data Control Modules

4.2.3.1 Data Control

There are two data control modules, shown in figures 4.7 and 4.8 with the only difference being the fact that one involves weights and the other does not. They are used in every layer and are the main computation block present in the overall design. ***Input*** is the current data point in the data stream. ***Weights*** are in the FMem. If more than one layer with weights is computing, the weights are in the same array, one after another. This is dealt with by calculating the total amount of weights present in the previous layers that are enabled and using that number as the start of the weights referring to the current layer. Both ***Offsets*** are calculated differently depending on which layer uses the module.

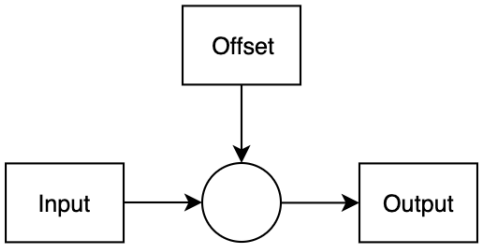


Figure 4.7: DataOffset module

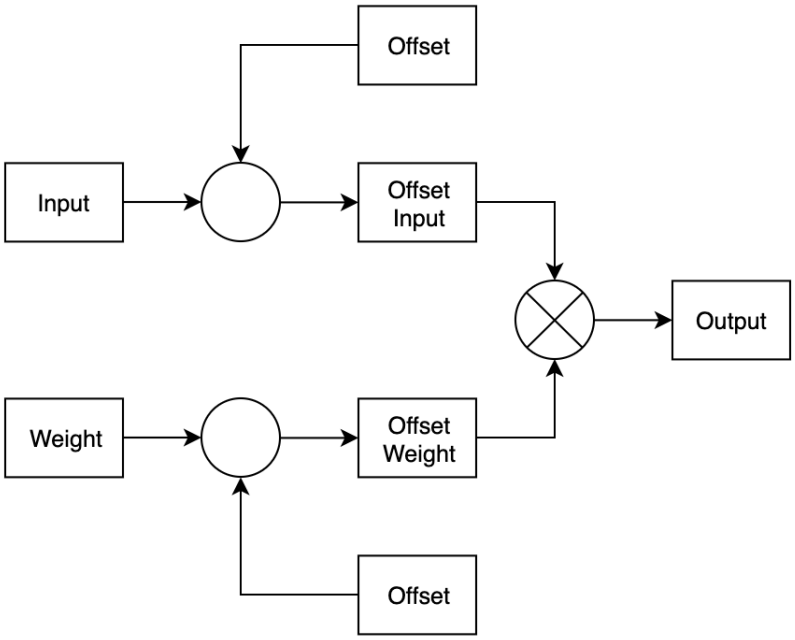


Figure 4.8: DataWeightOffset module

4.2.3.2 Input Control

Figure 4.9 shows the input data control module. ***Input*** is a value coming from the LMem, while ***DataInEnable*** is a simple enable signal and is calculated using different logic for each layer. ***a*** is used to accomplish different tasks, depending on the layer, mostly coming down to stream manipulation.

4.2.3.3 Output Control

The output data control module is shown in figure 4.10. ***Input*** is the computed value, for example, the output of matrix multiplication on fully connected layer. ***Activation function*** is configurable by the user and is set when configuring the network. ***ActEnable*** is set to 1 when the output value has been fully accumulated and is ready to be passed through the activation function. It's calculated differently depending on which layer uses the module. **PadEnable** is a simple enable signal used to deal with the LMem burst size, is only active after the last output point is calculated and is also calculated in a different way for each layer.

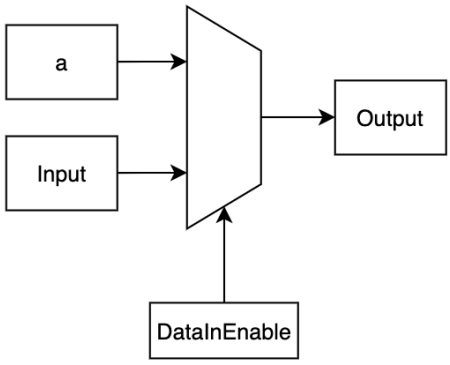


Figure 4.9: Input control module

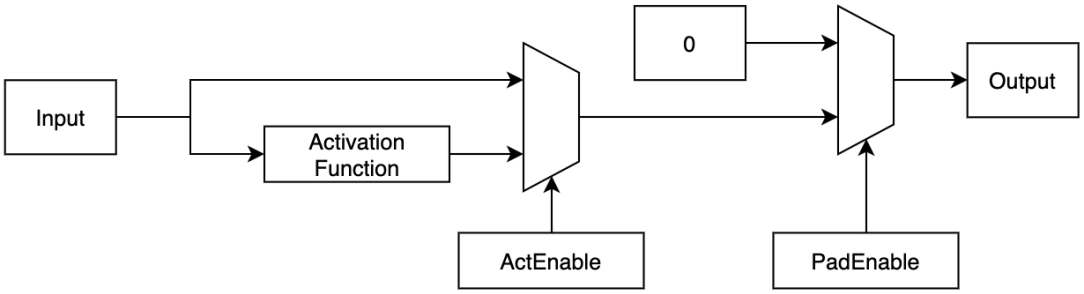


Figure 4.10: Output control module

4.2.4 Convolutional Layer Modules

4.2.4.1 Forward Propagation

When designing the forward propagation module, there are a few restrictions that must be considered. These are the LMem burst size and FMem dimensions. Furthermore, the design must include a way to customize computation resources.

Algorithm 1 Convolutional Layer Forward Propagation

Require:
In (NChannels × InDims × InDims)
Weights (NKernels × NChannels × KSize × KSize)
Ensure:
Out (NKernels × OutDims × OutDims)

```
1:  function CONVOLVE(In, W, inY, inX)
2:      Sum = 0
3:      for y ← 0:KSize do
4:          for x ← 0:KSize do
5:              Sum += In[inY + y][inX + x] * Weights[y][x];
6:          end for
7:      end for
8:      return Sum
9:  end function
10: function CONVOLUTION()
11:     for k ← 0:NKernels do
12:         for c ← 0:NChannels do
13:             for inY ← 0:InDims - KSize do
14:                 for inX ← 0:InDims - KSize do
15:                     Out[k][inY][inX] = Convolve(In[c], Weights[k][c], inY, inX);
16:                 end for
17:             end for
18:         end for
19:     end for
20: end function
```

By carefully analyzing algorithm 1, one can realize that there are only two options when it comes to running computation in parallel, lines 11 and 12. Assuming with no parallelism each output point is calculated in time t, the first option means calculating p points in time t, while the second means calculating one point in time $\frac{t}{p}$ - both options are equally good. However, calculating multiple points has disadvantages, one beeing that the calculated output would have to be summed with results already present in memory, because of the accumulation in line 5, requiring one additional LMem stream. As such, for this layer, the parallelism parameter, capped at $\frac{NChannels}{2}$, previously mentioned in section 4.2.1 determines how many channels are calculated in parallel, meaning $\frac{NChannels}{Parallelism}$ has to be an even number. Regarding the LMem burst size, this was resolved by adding one parameter from the CPU used to control when the layer outputs, be it real

data or padding. The restriction with FMem dimensions was dealt with by splitting computation, using at most two kernels each time the DFE is ran. As such, considering two $512 \times 7 \times 7$ kernels, an extreme case, means the FMem contains at most 50176 values, an amount well under the 65536 limit. The reason two were chosen instead of one is due to the burst size - if weights from only one kernel were available, padding would be necessary in between output channels, involving either complex logic when reading data in the next layers, require the CPU to read and re-write data correctly between the computation of each layer or make data in the LMem not be organized as shown in figure 4.6.

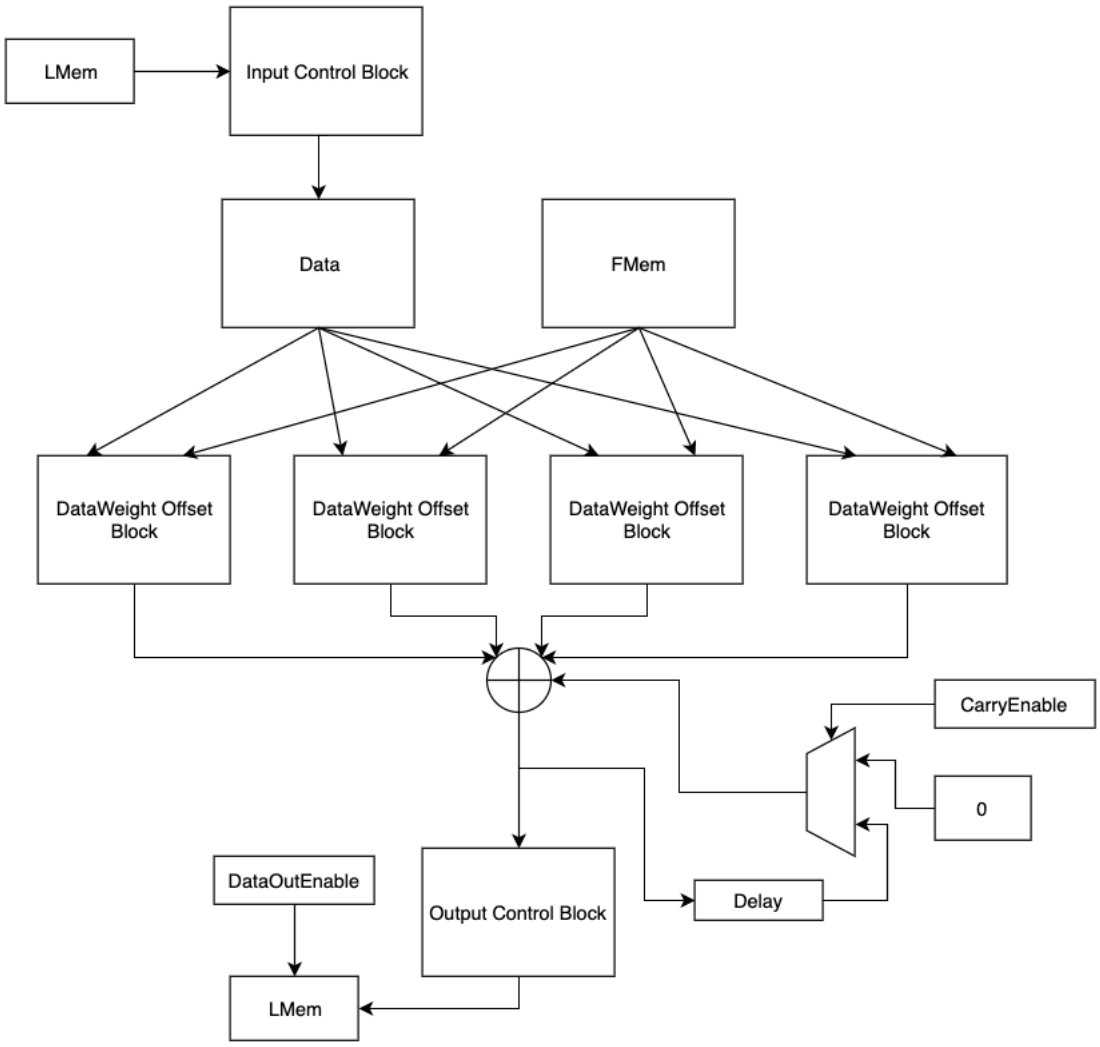


Figure 4.11: Conv layer FProp module

The module is composed of several simpler modules and components. Going from top to bottom, the input LMem stream is connected to an *input control block*. For this layer, ***a*** is used in conjunction with ***DataInEnable*** to handle both types of padding. LMem padding is controlled by using one counter to disable computation on ticks where the memory input is padded and

parameter padding is controlled by making $a = 0$ and using nested counters *CurY* and *CurX*, which contain the value of the current input position. If the input position is within bounds, data is read from LMem, otherwise, the enable signal is set to 0 and the output is set to 0.

The data is then connected into a customizable amount of *DataWeightOffset* modules given by equation 4.2. The weights are configured by the CPU, accessed using the FMem and are different for each DFE run, changing as each kernel finishes computing. The offsets are calculated as shown in equations 4.3 and 4.4. Kernel is either 0 or 1, since, as mentioned above, at most 2 kernels are computed per DFE run. Channel, y and x are nested counters, exactly as algorithm 1. The only difference is the increment of channel is $\frac{NChannels}{Parallelism}$. As an example, if $NChannels = 8$ and $Parallelism = 4$, channels 0,2,4,6 are calculated in parallel, followed by channels 1,3,5,7 - computation is finished as soon as channel 1 is calculated. Each $(Channel, y, x)$ combination goes into one of the modules.

The output from every module is summed and the result is fed into both a carry circuit and an *output control block*. The carry circuit, is composed of one delay node set to $(InDims + 2 * Padding)^2$ and one multiplexer used to filter false carried values when computing the first channel. Its role is to carry computation results from previous ticks, more specifically, the previous channel, as evidenced by the delay node. The output control module works exactly as described in section 4.2.3, with **ActEnable** being set to the same as **DataOutEnable**, because any time the module writes to the LMem, computation for each output point is already finished. The output data is then written to the LMem, when *DataOutEnable* is set to 1. Output control is achieved by using counters which control the input position, updating every tick, and one CPU parameter, **FirstOutput**, described further in 4.2.9. During the last iteration, padding is started as soon as the last output point is calculated, so the output stream is aligned faster. Furthermore, stride control is accomplished by only considering every *nth* output point, where *n = stride* and discarding the rest.

$$\text{NConvDataWeightOffsetModules} = \text{Parallelism} \times KernelSize^2 \tag{4.2}$$

$$\begin{aligned} \text{DataOffset} = &\text{Channel} \times \text{InDims}^2 + \\ &y \times \text{InDims} + x \end{aligned} \tag{4.3}$$

$$\begin{aligned} \text{WeightOffset} = &\text{Kernel} \times \text{InChannels} \times \text{KernelSize}^2 + \\ &\text{Channel} \times \text{KernelSize}^2 + \\ &y \times \text{KernelSize} + x \end{aligned} \tag{4.4}$$

The module uses only two LMem streams and has to be ran $\left\lceil \frac{\text{NKernels} \times \text{OutDims}^2}{\text{BurstSize} \times \text{BurstMult}} \right\rceil$ times for the entire output to be calculated.

4.2.4.2 Backward Propagation

The restrictions faced are the same as in the forward propagation module, with the addition of trying to minimize LMem streams, to allow for the maximum number of layers in each block, because two operations are required, the error calculation and the weight update.

Algorithm 2 Convolutional Layer Backward Propagation

Require:
FwdOut (NKernels × OutDims × OutDims)
Error (NKernels × OutDims × OutDims)
FwdIn (NChannels × InDims × InDims)
Weights (NKernels × NChannels × KSize × KSize)

Ensure:
Out (NKernels × OutDims × OutDims)

```
1:  function CALCERROR()                    ▷ Same as FProp, with kernels and channels switched
2:      for c ← 0:NChannels do
3:          for inY ← Padding:DeltaDims - KSize - Padding do
4:              for inX ← Padding:DeltaDims - KSize - Padding do
5:                  for k ← 0:NKernels do
6:                      Out[c][inY][inX] += Convolve(Delta[k], Weights[k][c], inY, inX);
7:                  end for
8:              end for
9:          end for
10:     end for
11: end function
12: function UPDATEWEIGHTS()        ▷ Same as a Normal convolution, using Delta as Weights
13:     for k ← 0:NKernels do
14:         for c ← 0:NChannels do
15:             for inY ← 0:PrevInDims - DeltaDims do
16:                 for inX ← 0:PrevInDims - DeltaDims do
17:                     W[k][c][y][x] -= η ×Convolve(FwdIn[k], Delta[k][c], inY, inX);
18:                 end for
19:             end for
20:         end for
21:     end for
22: end function
23: function CONVBACKPROP()
24:     Delta = Error × f'(FwdOut)      ▷ Element Wise pass through Act Func derivative
25:     CalcError();                        ▷ Calculate Error to backprop onto next layer
26:     UpdateWeights();                    ▷ Update Weights. KSize is now DeltaDims.
27: end function
```

Analyzing algorithm 2 reveals one big problem - two convolutions are required, which would use a total of 4 streams if the forward propagation module were used for both. Since there are 3 total inputs required for both convolutions, this would total 7 LMem streams, which would mean each block could have no more than one convolution layer. Furthermore, since the weight

update convolution uses delta as the kernels, the total amount of weights would exceed the FMem dimensions and the DFE would not have enough multipliers to perform so many multiplications. As such, this approach is not feasible.

Taking all these problems in consideration, a new module was designed. Before its introduction, the weight update block will be presented, as it constitutes a large part of the overall module.

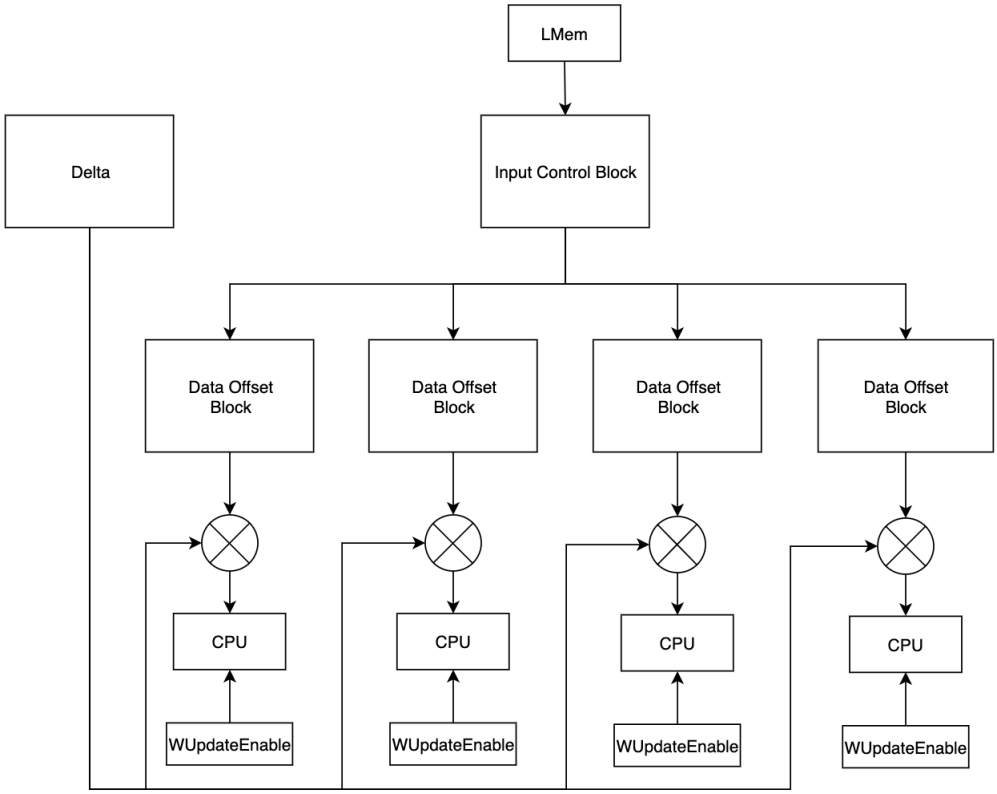


Figure 4.12: Conv layer WeightUpdate module

This module computes the weight updates not directly as a convolution, because, as said above, the DFE does not have enough resources to do so. Instead, the update for each weight referent to one delta position is calculated. The sum of all updates is the final result. To avoid one extra LMem stream, each part of the weight update calculation is streamed directly to the CPU as a vector of size $KSize^2$. The vector has an enable stream **WUpdateEnable**, which simply filters out incorrect results, streaming zero to the CPU when not enabled, because of the fact that the CPU will sum all updates to the weights, meaning wrong values would result in incorrect updates. The amount of DataOffsetBlocks is $KSize^2$.

The overall module is described in figure 4.13. There are two LMem data streams at the top, which contain **FwdOutput** and **Error**. They are both fed into *input control blocks*, to control padding. Similarly to forward propagation, this is controlled by making $a = 0$ and using DFE counters to control the input position. The activation function block simply calculates delta, exactly as in algorithm 2.

The delta value is fed into one *WeightUpdate Block*, described above, and a chain of blocks responsible for the convolution calculation, which works almost exactly as forward propagation, the only difference beeing offset calculation, which simply switches *InDims* for *DeltaDims* because computation is done with input and output dimensions flipped and switches *Channel* for *Kernel*, because the innermost loop is now done per output channel, instead of input channel, which also implies parallelism is now capped at $\frac{NKernels}{2}$. In addition, the module does not calculate the activation function before outputting, because it's calculated before computation on the next layer.

The module uses four LMem streams and has to be ran $\left\lceil \frac{\text{NChannels} \times \text{InDims}^2}{\text{BurstSize} \times \text{BurstMult}} \right\rceil$ times for the entire output to be calculated.

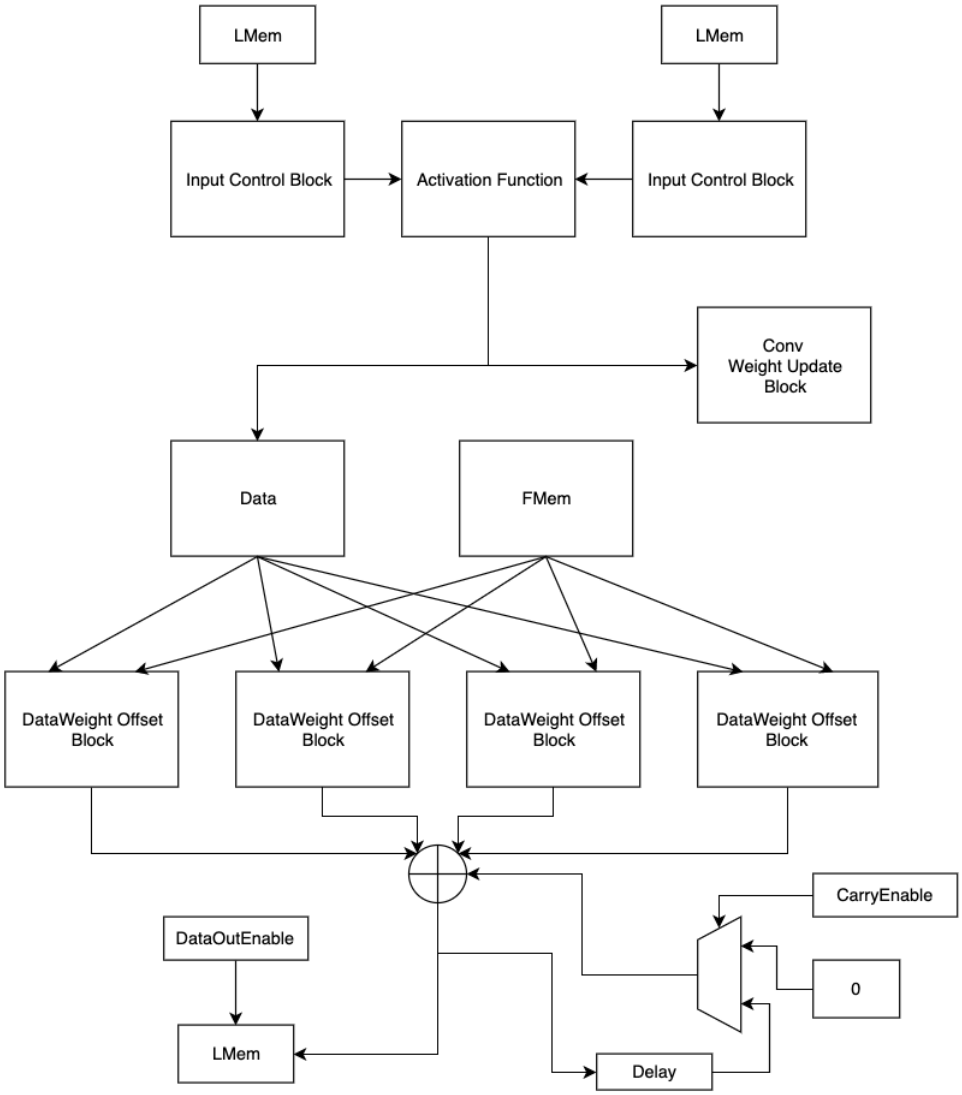


Figure 4.13: Conv layer BProp module

4.2.5 Pooling Layer Modules

4.2.5.1 Forward Propagation

When designing the pooling module, the only restriction is the burst size, handled by outputting values from different channels, similarly to the convolution module. Both max pooling and mean pooling are shown in algorithm 3.

Algorithm 3 Pooling Layer Forward Propagation

Require:
In (NChannels × InDims × InDims)
Wsize

Ensure:
Out (NChannels × OutDims × OutDims)

```
1:  function MeanPoolWindow(In, c, inY, inX)          ▷ Calculate average of window
2:      Sum = 0
3:      for y ← 0:WSize do
4:          for x ← 0:WSize do
5:              Sum += In[c][inY + y][inX + x];
6:          end for
7:      end for
8:      return Sum/WSize^2
9:  end function
10: function MaxPoolWindow(In, c, inY, inX)           ▷ Calculate max value inside window
11:     Max = -∞;
12:     for y ← 0:WSize do
13:         for x ← 0:WSize do
14:             if In[c][inY + y][inX + x] > Max then
15:                 Max = In[c][inY + y][inX + x];
16:             end if
17:         end for
18:     end for
19:     return Max
20: end function
21: function Pool()                                   ▷ Calculate Out
22:     for c ← 0:NChannelsSize do
23:         for inY ← 0:InDims - WSize do
24:             for inX ← 0:InDims - WSize do
25:                 Out[c][inY][inX] = PoolWindow(In, c, inY, inX);   ▷ Either Max or Mean
26:             end for
27:         end for
28:     end for
29:     return Out
30: end function
```

Since there are no accumulations, this module was chosen not to be parallelized, because the only option would be calculating multiple points per tick, requiring multiple LMem streams.

The overall architecture for max pooling is shown in figure 4.14. For mean pooling, the only different component is the max calculator, which is changed for a sum and divide.

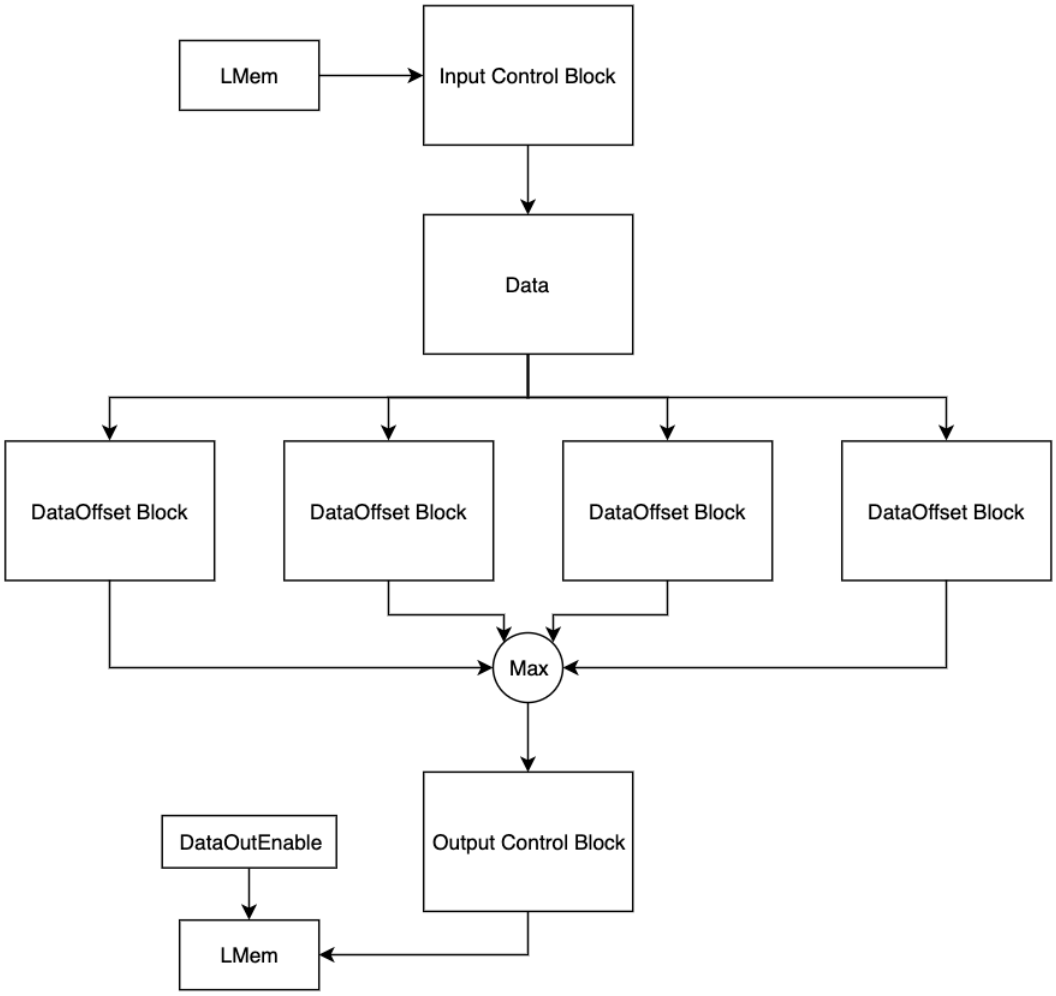


Figure 4.14: Pooling layer FProp module

$$\text{DataOffset} = \text{y} \times \text{InDims} + \text{x} \tag{4.5}$$

$$\text{NPoolDataOffsetModules} = \text{WindowSize}^2 \tag{4.6}$$

Going from top to bottom once again, the input data stream incoming from the LMem is fed into an *input control block*. The value of *a* doesn't matter, since *DataInEnable* is only used to control when the layer is computing or not, as no other control is necessary, because padding was not implemented for the pooling layer.

The data then goes into an amount of *data offset blocks* given by equation 4.6, to select the pooling window from the input stream. Once again, the offset calculation uses nested counters *y*

and x, exactly as in algorithm 3, with each combination y, x used for each module. Each point is then fed either into a *max* node or an *average* node, depending on the pooling type.

The *output control block* works the exact same way as in the convolutional layer (section 4.2.4.1). *DataOutEnable* controls the layer output by using two counters, which update every tick, to control the input position and the same CPU parameter, *FirstOutput*, previously mentioned in section 4.2.4.

The module uses two LMem streams and has to be ran $\left\lceil \frac{\text{NChannels} \times \text{OutDims}^2}{\text{BurstSize} \times \text{BurstMult}} \right\rceil$ times for the entire output to be calculated.

4.2.5.2 Backward Propagation

Similarly to forward propagation, the only design restriction is the burst size, handled in the exact same way. Regardless of the type of pooling the module stays the same, because of the fact that values in the mask calculated during forward propagation depend on the type used.

Algorithm 4 Pooling Layer Backward Propagation

Require:
Delta (NChannels × OutDims × OutDims)
Mask (NChannels × InDims × InDims)
WSize

Ensure:
Out (NChannels × InDims × InDims)

```
1:  function MaskMult(c, Y, X)                          ▷ Multiply by mask
2:      Sum = 0
3:      for y ← 0:WSize do
4:          for x ← 0:WSize do
5:              MaskY = Y*WSize + y;
6:              MaskY = Y*WSize + x;
7:              Out[c][MaskY][MaskX] = Mask[c][MaskY][MaskX] * Delta[c][Y][X][
8:          end for
9:      end for
10: end function
11: function Pool()                                     ▷ Calculate Out
12:     for c ← 0:NChannels do
13:         for Y ← 0:OutDims do
14:             for X ← 0:OutDims do
15:                 Out[c][inY][inX] = MaskMult(c, Y, X);
16:             end for
17:         end for
18:     end for
19:     return Out
20: end function
```

For the same reasons as forward propagation, this module doesn't have parallelism.

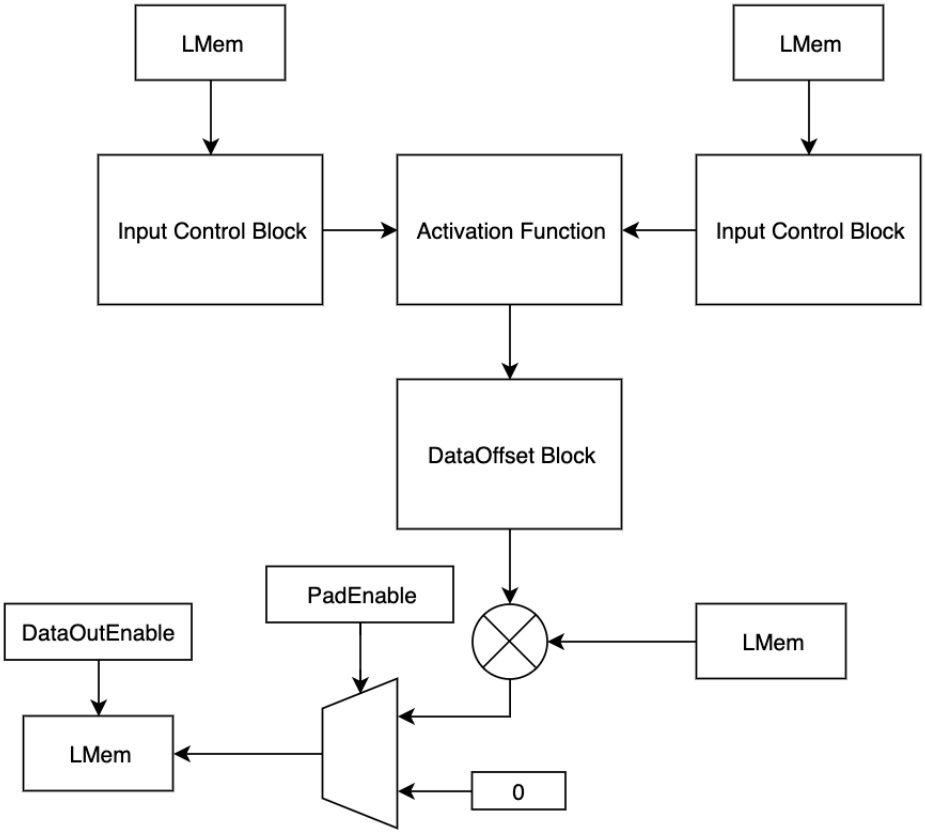


Figure 4.15: Pooling layer BProp module

$$\text{DataOffset} = \text{YStrideTicks} \times \text{InDims} \tag{4.7}$$

There are two input data stream incoming from the LMem, the output from forward propagation and the error from the next layer. Both are used to calculate *delta*. Since both have the same dimensions, the calculation is simply $Error \times f'(Output)$, element wise. The value of a doesn't matter, because, in this module, *DataInEnable* is only used to control stride.

Delta then goes into a *data offset block*. The offset calculation is shown in equation 4.7, which simply selects the correct delta value for the mask window. **YStride** is a counter which takes values from $0 : \text{Stride}$, depending on which row the input data stream is currently at. The offset value is then multiplied by the Mask value, which is also present in the LMem.

Output control is achieved using **DataOutEnable** to not only make sure wrong values aren't written into memory, but also to align the output stream and.**PadEnable** simply changes the output value to 0 when padding.

The module uses four LMem streams and has to be ran $\left\lceil \frac{\text{NChannels} \times \text{InDims}^2}{\text{BurstSize} \times \text{BurstMult}} \right\rceil$ times for the entire output to be calculated.

4.2.6 Fully Connected Layer Modules

4.2.6.1 Forward Propagation

The restrictions faced when designing the fully connected layer are the LMem burst size and FMem dimensions. Similarly to the convolutional layer, a suitable way to customize computation resources also has to be developed. The algorithm used in the fully connected layer is matrix multiplication, between the input as weights, which is calculated as follows:

Algorithm 5 Matrix Multiplication

Require:
In (InDims)
W(InDims $\times$ OutDims)
Ensure:
Out (OutDims)

```
1: function MATRIXMULTIPLY()                          ▷ Calculate Out
2:     for y ← 0:OutDims do
3:         for x ← 0:InDims do
4:             Out[y] += In[x] * W[x][y];
5:         end for
6:     end for
7: end function
```

Before discussing parallelism strategies, FMem dimensions must be discussed, as fully connected layers have a lot more weights than convolutional layers. Ideally, the weights for every input point and burst size output points would be loaded - only two streams would be required. However, considering VGG16 as an example network quickly rules this option out. The pooling layer before the fully connected layer has output dimensions $512 \times 7 \times 7$. Considering *BurstMult = 1*, burst size is 24. This means the FMem would have to contain a total of 600000 weights, a value well above the 65535 limit in the weight array. As such, a different solution was adapted - burst size is considered for both the input and the output, resulting in a total amount of weights equal to $(\text{BurstMult} \times \text{BurstSize})^2$. This comes at the cost of one additional LMem stream, making this module require one more than other layers. Furthermore, additional output logic is required, to add the computed value to the one already in memory, since only part of the output is beeing calculated each time the kernel is ran. This is achieved by using the CPU parameter, *First Output* already mentioned previously to control the input memory stream position.

Regarding parallelism, there are, again, two options, the inner loop or the outer loop. For the same reason presented in the convolutional layer (section 4.2.4.1), the inner loop was chosen. Therefore, for this layer, the parallelism parameter, capped at $\frac{\text{BurstMult} \times \text{BurstSize}}{2}$, determines how many inputs are used per DFE run. For example, if $BurstMult = 1$ and $Parallelism = 12$, for each of the 24 outputs, inputs 0,2,4,6,8,10,12 are used in parallel, followed by inputs 1,3,5,7,9,11,13.

As an added bonus, the LMem burst size issue is automatically resolved, because of the fact that every LMem stream present in the module has its size defined in multiples of the burst size.

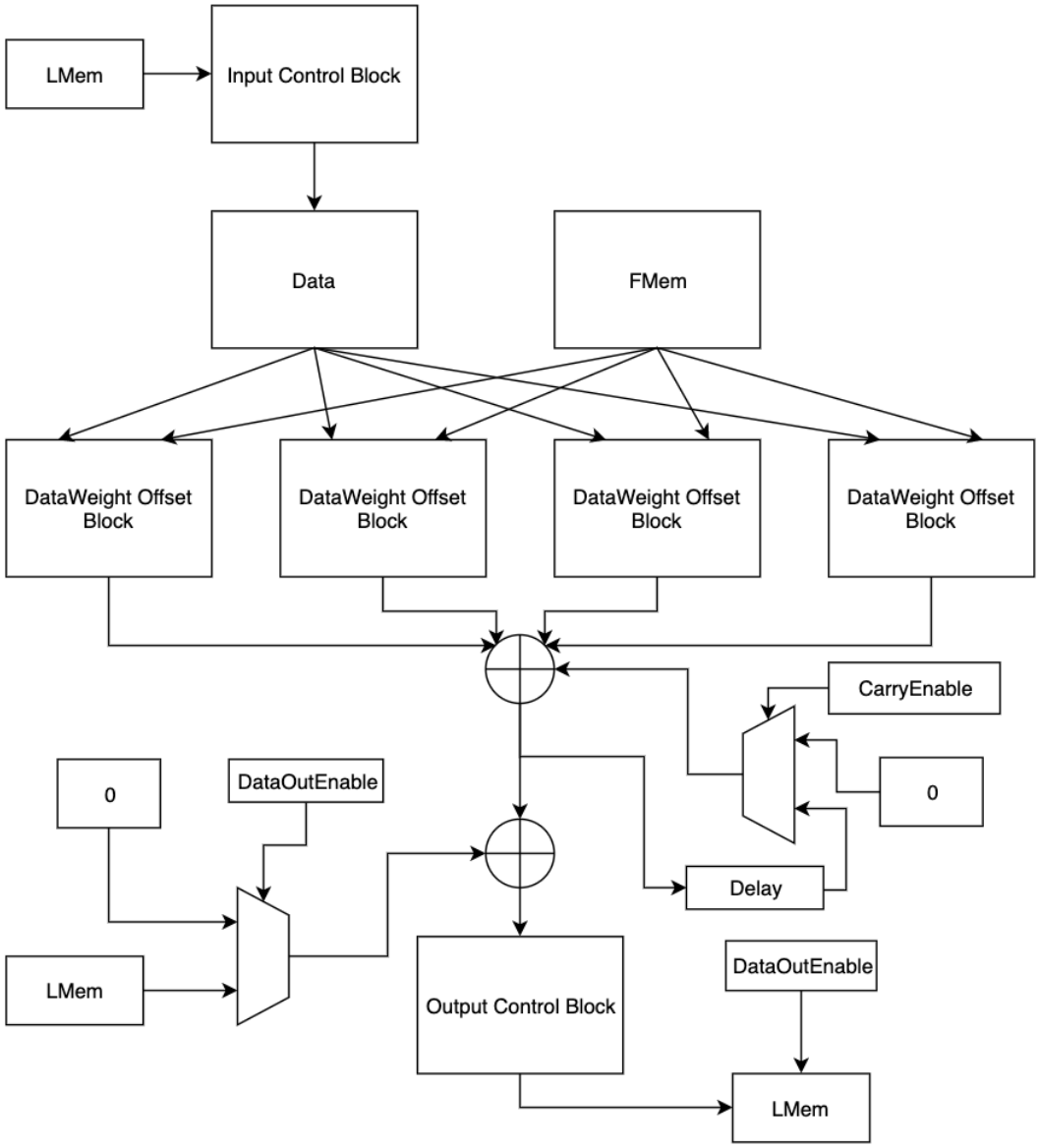


Figure 4.16: Fully connected layer FProp module

The overall architecture is shown in figure 4.16. Going from top to bottom once again, the input data stream incoming from the LMem is fed into an *input control block*. The value of a is the previous value from the input stream, since, for each input stream point, each output point needs to be calculated before moving on to the next input stream point. *DataInEnable* is used for this exact purpose and is calculated by using an output counter, which takes values from 0:(BurstMult$\times$BurstSize), the size of the output stream.

The data then goes into a configurable amount of *data weight offset blocks* given by equation 4.8, to select points from the input stream. The data offset calculation, shown in equation 4.9 is simple, selecting input point *CurIn* from the stream. The weight offset, shown in equation 4.10 is also simple, selecting an input point in the same way as data offset and selecting the output point

by simply summing the index of the current output point.

The data is the multiplied by the corresponding weight and all the outputs are summed. This is then forwarded into the same carry circuit as described in the convolutional layer (section 4.2.4). Delay is instead set to BurstSize $\times$ BurstMult, to carry results from the previous input point. The output from the previous run, already present in memory is summed, but only when calculation for the current run is complete. The *output control block* works exactly as the other layers, with **DataOutEnable** beeing controlled by the *CurIn* counter, which contains the index of the current input. When input $\frac{\text{BurstMult} \times \text{BurstSize}}{\text{Parallelism}}$ is reached, computation is done and the signal is set to one.

$$\text{NFconDataWeightOffsetModules} = \text{Parallelism} \tag{4.8}$$

$$\text{DataOffset} = \text{CurIn} \times \text{BurstSize} \times \text{BurstMult} \tag{4.9}$$

$$\text{WeightOffset} = \text{CurIn} \times \text{BurstSize} \times \text{BurstMult} + \text{CurOut} \tag{4.10}$$

The module uses three LMem streams and has to be ran $\left\lceil \frac{\text{InDims} * \text{OutDims}}{(\text{BurstSize} \times \text{BurstMult})^2} \right\rceil$ times for the entire output to be calculated.

4.2.6.2 Backward Propagation

The restrictions taken in mind when designing backpropagation are the same as forward propagation, with the addition of trying to minimize LMem streams.

Algorithm 6 FconBackProp

Require:
FwdOut (OutDims)
Error (OutDims)
FwdIn (InDims)
Weights (InDims $\times$ OutDims)

Ensure:
Out (InDims)

```
1: Delta = Error × f'(FwdOut)          ▷ Element Wise pass through Act Func derivative
2: for x ← 0:InDims do
3:     for y ← 0:OutDims do
4:         Out[x] += Delta[y] * W[x][y];                    ▷ Calculate Output
5:         W[x][y] -= η × Delta[y] × FwdIn[x]               ▷ Update Weights
6:     end for
7: end for
```

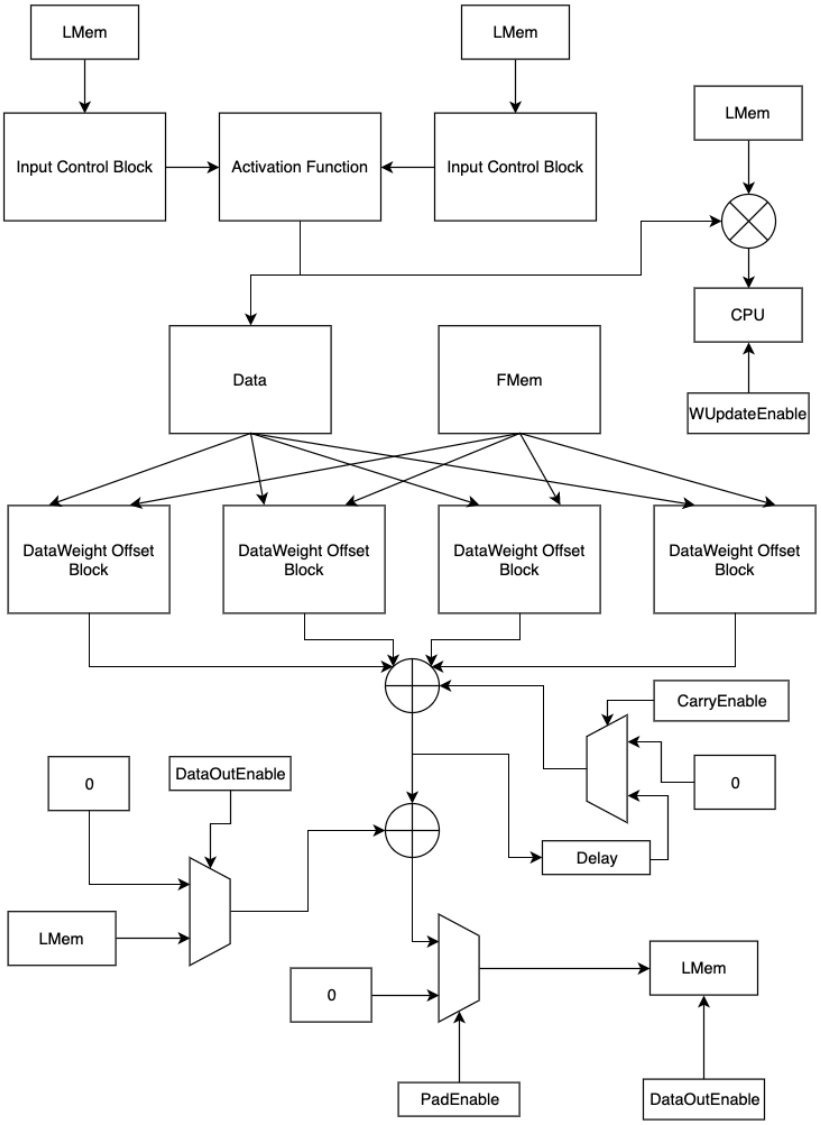


Figure 4.17: Fully connected layer BProp module

Similarly to backpropagation on other layers, there are two LMem streams containing *FwdOut* and *Error*, which are used to calculate *Delta*.

This value is then multiplied by *FwdIn*, present in the LMem. The corresponding value represents the weight update and is streamed directly to the CPU, to avoid using another LMem stream. The remaining blocks are used to calculate the output, in the same way as forward propagation, with a few differences regarding offset calculation - input and output counters are flipped, because input and output dimensions are flipped. No activation function block is necessary, for the same reasons previously explained in other backpropagation modules.

The module uses five LMem streams and has to be ran $\left\lceil \frac{\text{InDims} \times \text{OutDims}}{(\text{BurstSize} \times \text{BurstMult})^2} \right\rceil$ times for the entire output to be calculated.

4.2.7 Kernel

Kernels are responsible for computation, run for a certain number of ticks configured in the manager and must be designed in accordance with resource limitations described in section 4.1.1.

4.2.7.1 Bitwidth Configuration

When performing computation on a DFE, a number representation system must be chosen. In compliance with the defined objectives, the bitwidth can be set by the user, with two restrictions beeing in place - only *floating point systems* are allowed and *rules in table A.1*, which are implicit by maxeler must be followed.

There are several reasons choosing floating point versus fixed point, which include the fact that neural networks are very sensitive to small changes, so high computation accuracy is desired. Furthermore, the design becomes simpler when only one number system is chosen and the maxeler API is considerably simpler for floating point systems.

4.2.7.2 Activation Function Calculation

Since the weights are initialized as mentioned in 2.2, overflow from multiplications and accumulations should not be a problem. However, in certain training scenarios, when using ReLu, depending on network architecture and used bit width, this output might overflow. Therefore, a threshold T was implemented, limiting the output value, which was set to 10, because, from experimentation with the C API, it was attested that computation values seldom exceed this value.

Regarding sigmoid and TanH, these functions involve calculation with the exponential function. Maxeler provides a math API which allows for simple e^x computation, but, once again, overflow problems can occur. One possible solution is presented in [14], which defines computational regions for the TanH function - once x is larger than $12.5ln(2) \approx 8.66$ (for 32 bit floats), computation can be skipped. Since TanH uses e^{2x}, this result means e^x computation can be skipped whenever $x >= 17.32$. This can be extended to the sigmoid function, which requires e^{-x}, so computation is skipped if $x < 17.32$, countering any overflow that might occur.

Concluding, the hardware calculation for each of the implemented activation functions is presented below:

$$(ReLu) \quad f(x) = \begin{cases} 0, & x \leq 0 \\ \text{min(x,10)}, & x > 0 \end{cases} \tag{4.11}$$

$$(Sigmoid) \quad f(x) = \begin{cases} 0, & x \leq -17.32 \\ \frac{1}{1+e^{-x}}, & x > -17.32 \end{cases} \tag{4.12}$$

$$(Tanh) \quad f(x) = \begin{cases} 1, & x \geq 8.66 \\ 2 \times (0.5 - \frac{1}{e^{2x}+1}), & x < 8.66 \end{cases} \tag{4.13}$$

4.2.7.3 Forward Propagation

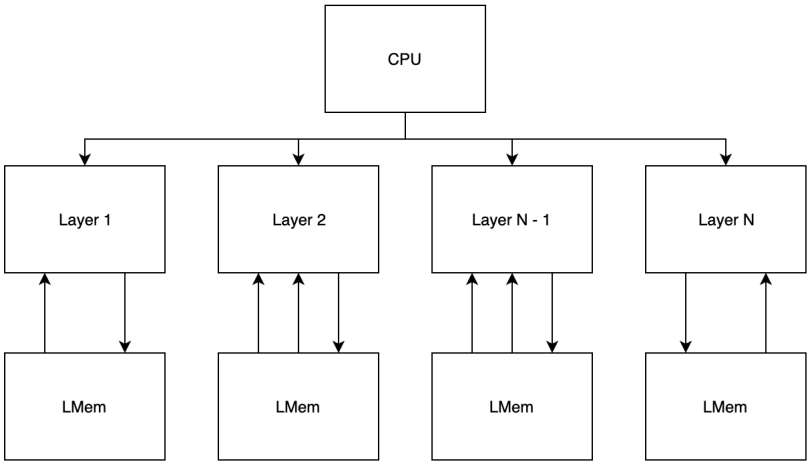


Figure 4.18: Kernel FProp module

As evidenced in figure 4.18, every layer runs in parallel. Naturally, computation can only occur if each layer is enabled and correct input data is present in the respective input stream. This information is streamed from the CPU in the form of one parameter, **MemControl**, which the kernel uses as an enable signal. Data correctness is implicit, due to the fact that the parameter is automatically to 0 by the CPU for DFE runs where the data present in the LMem is not correct.

4.2.7.4 Backward Propagation

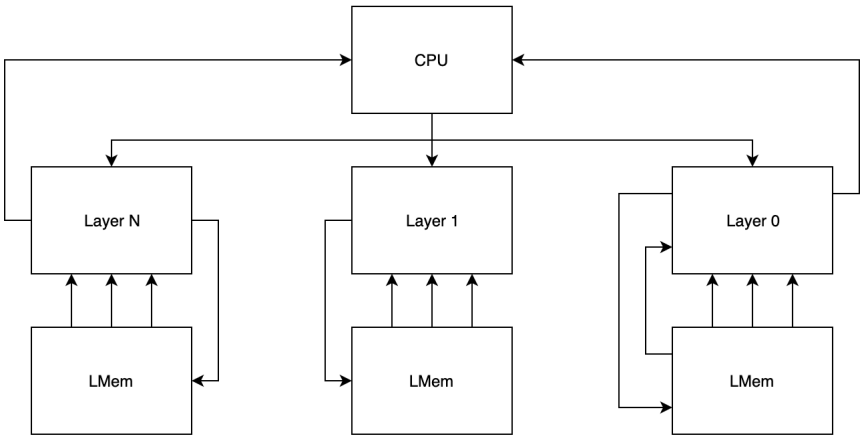


Figure 4.19: Kernel BProp module

The backpropagation module is very similar to the forward propagation module. The only two differences are the extra CPU connection and the fact that layers are ran in backward order.

4.2.8 Manager

The Manager configures the entire DFE dataflow, be it LMem streams, FMem or CPU streams, as well as clock frequency, LMem access frequency and also generates interfaces, which can then be called by the CPU to run the Kernel.

4.2.8.1 Customization

Before addressing anything else, in order to make both the network architecture and parallelism options fully customizable, there has to be a way for the manager to access these values at compile time. However, the manager is always compiled before the CPU, so the respective interfaces can be created for the CPU to use. As such, a way for the manager to have access to these parameters must be devised.

From digging through the maxeler documentation, this is apparently not possible, which would mean multiple modules would have to be created for each desired combination of parameters, an unfeasible approach. A workaround was found however, by implementing a scheme which consists of writing the chosen network configuration to a file in the CPU code and reading that file before anything else is done in the manager. The end result means running the CPU code once, with the chosen network configuration, compiling the manager and running the CPU code a second time, with the manager interface now correctly configured. There are 20 files, 10 for forward propagation and 10 for backward propagation, each reading the block configuration file prepared by the CPU and creating one block, if the file exists, with respective read, write and run interfaces.

There is only 1 kernel, which runs all layers of a block every tick, depending on the **MemControl** signal streamed from CPU, as mentioned in section 4.2.7. This proved to be the simplest approach, versus running multiple kernels, each running a single layer, which would require much more complex interface and dataflow configurations.

Furthermore, using the single kernel approach, LMem streams could be multiplexed in order to allow very deep networks to be compiled entirely in one block, albeit with minimal parallelism, depending on the available hardware. This strategy is implemented by default in [3] in order to increase single layer performance, not to allow for deep networks.

This was not explored because the parallelism parameter already allows for hardware resource configuration and yields better results versus compiling deep networks in one block. This is also the reason why there are specific blocks for forward propagation and backward propagation. The maximum amount of LMem streams is 15, where 2 are used by Write and Read, which are required on every block. If both types of propagation were configured, the maximum amount of layers in a block would be 2, largely invalidating the block architecture approach. Another unwanted consequence of this stream multiplexing strategy is that fact that the C API call would change depending on how many layers are present, because extra parameters are necessary from the CPU to be used as the select parameter for the stream multiplexers.

4.2.8.2 Interfaces

Each block has 3 total interfaces, created and configured by the manager:

1 - Write

Uses one LMem stream for writing data from the CPU directly to the LMem. Required in all Blocks, because the LMem resets every time a new block is loaded. There are two required parameters, the position to begin writing in and the size, which must be managed by the CPU in order to comply with the Burst Size.

2 - Read

Uses one LMem stream, for reading data from the LMem and streaming it directly to the CPU. Required in all Blocks, because the LMem resets every time a new block is loaded. There are two required parameters, the position to begin reading from and the size, which must be managed by the CPU in order to comply with the Burst Size.

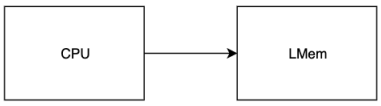


Figure 4.20: LMemWrite interface

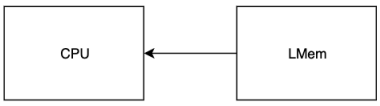


Figure 4.21: LMemRead interface

3 - Run

Runs either forward propagation (figure 4.22) or backward propagation (figure 4.23), depending on the block.

Uses a variable amount of LMem streams, with the maximum beeing 13, because 2 are used by Write and Read. Stream management is done automatically without any user input, with two required parameters, streamed directly from the CPU, **MemControl** and **First Output**. **MemControl** serves as both an enable and as output memory control, so that values are written into the correct place in the LMem. **First Output** is only used in the manager on fully connected layers, as input memory control, so correct inputs are read. This is not necessary in convolutional or pooling layers due to the fact that computation requires the entire input volume. The Kernel runs for a certain number of ticks, also using both parameters, as mentioned in section 4.2.7. This value is set to the maximum possible amount, but doesn't reflect the actual amount of ticks the kernel runs for, because computation stops as soon as every output data stream is aligned. For example, if one output point is written every tick and 24 points need to be written, the kernel only runs for 24 ticks, despite beeing configured to run for more.

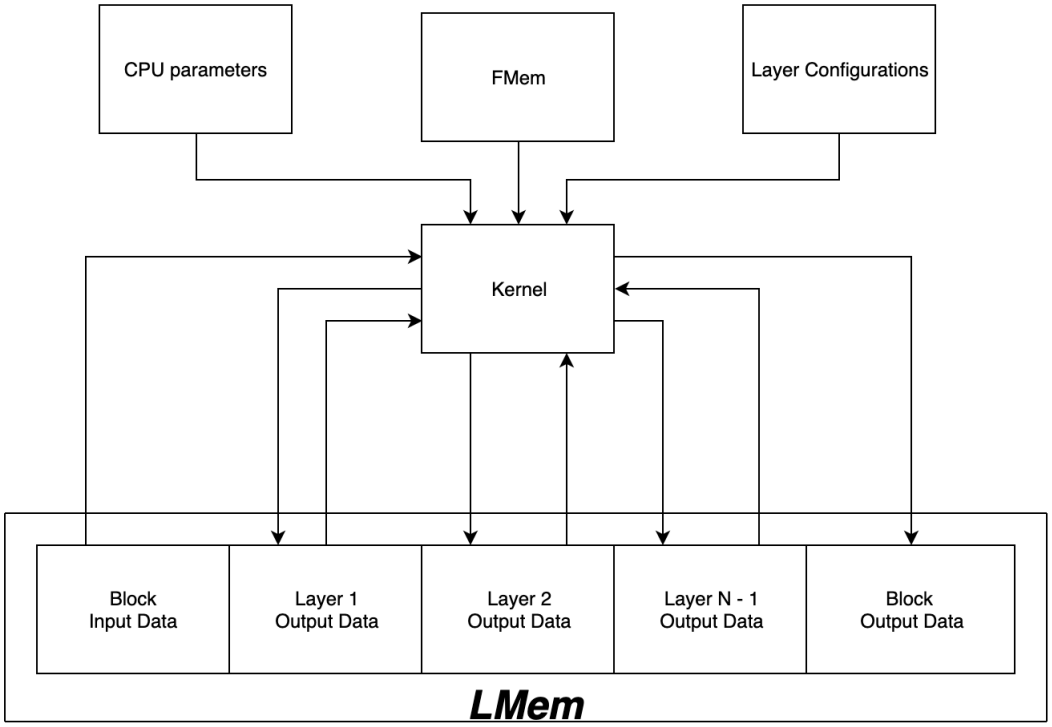


Figure 4.22: FProp Interface

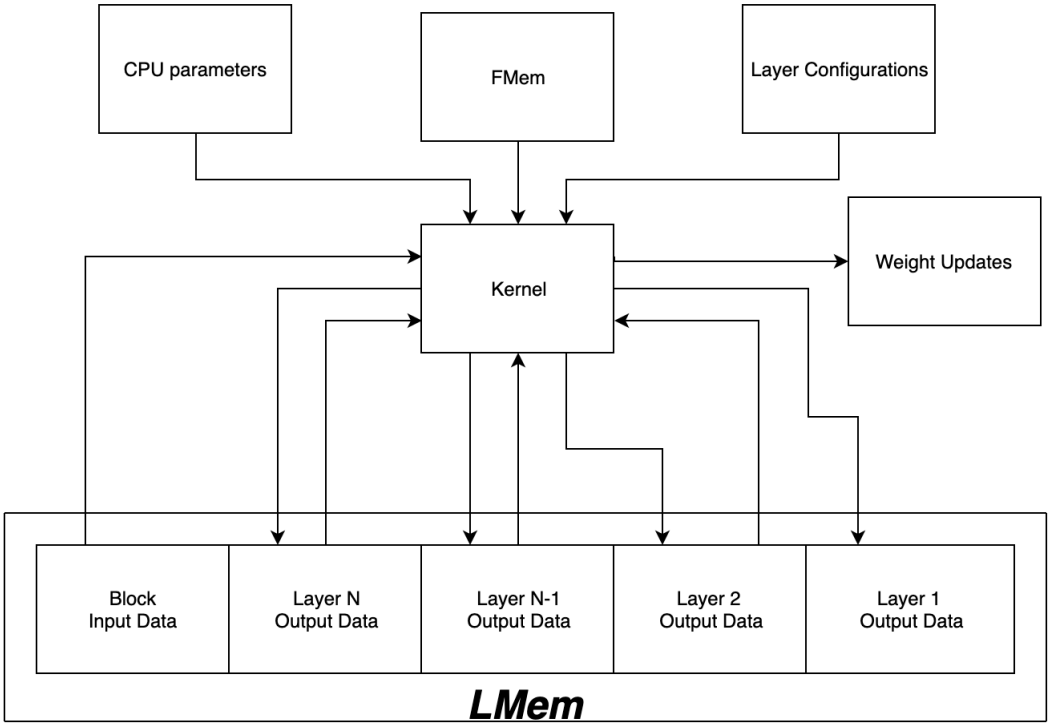


Figure 4.23: BProp Interface

4.2.9 CPU

The CPU is responsible for network creation, DFE parameter configuration and overall model control.

4.2.9.1 Network Creation

As mentioned previously, an API was written, to allow for a customizable network architecture. Listing 4.1 shows an example of a very shallow network. Parameters like the *Learning Rate*, *BatchSize* and *ErrorFunction* are set with specfic functions, described in A. If, at any point in the design, invalid parameters are selected, the API informs which layer had wrong parameters and why those are invalid.

```
void CreateTestNetwork(Net)
{
  int InDims[3] = {3, 28, 28};

  // Init Network
  InitCNN(Net, InDims);

  // First Block
  AddBlock(Net);
  AddConv(16, 3, 1, 1);
  AddActi(ReLu);
  AddConv(16, 3, 1, 1);
  AddActi(ReLu);
  AddPool(2, MaxPool, 2);

  // Second Block
  AddBlock(Net);
  AddFcon(1000);
  AddActi(Sigmoid);
  AddDrop(0.5);
  AddFcon(10);
  AddActi(Soft);

  // Frequency
  SetDesignFreq(150);
}
```

Listing 4.1: Network Creation example using the C API

4.2.9.2 DFE Setup

Just like network creation, the created API makes DFE setup extremely simple (listing 4.2). The user must provide values for *BurstMult* and *Parallelism*, for forward and backward propagation. These values are automatically written to specific files, which are read by the manager when blocks

are compiled to hardware and are also used by the CPU in order to automatically setup the required parameters by the kernel and manager.

```
void SetupDFE()
{
  Network* Net = malloc(sizeof(Network));

  CreateTestNetwork(Net);

  // Burst Multipliers
  int** FBM = malloc(Net->TotalBlocks * sizeof(int*));
  int** BBM = malloc(Net->TotalBlocks * sizeof(int*));

  // Parallelism
  int** FP = malloc(Net->TotalBlocks * sizeof(int*));
  int** BP = malloc(Net->TotalBlocks * sizeof(int*));

  for(int i = 0; i < Net->TotalBlocks; ++i)
  {
    int BS = Net->Blocks[i].BlockSize;

    FBM[i] = malloc(BS * sizeof(int));
    BBM[i] = malloc(BS * sizeof(int));
    FP[i] = malloc(BS * sizeof(int));
    BP[i] = malloc(BS * sizeof(int));
  }

  // Setup parameters as desired
  FBM[0][0] = 15;
  FP[0][1] = 8;

  // Configure Require parameters for Kernel computation
  DFECompile(Net, FBM, BBM, FP, BP);
}
```

Listing 4.2: DFESetup example using the C API

After running the CPU code once like this, the user is then free to compile any number of blocks for forward or backward propagation and run one of two functions, ***DFEInference()*** or ***DFETrain()***, which do exactly what is expected. Usage examples are shown in chapter A.

Parameter setup is done automatically when the user calls *DFECompile*. Before any parameters are calculated, the amount of times the kernel needs to be called for each block run is calculated. This value is always less than the sum of required calls for each layer, because layers start computing as soon as data becomes available. For each call where one or more layers are active, the following parameters are calculated: (c - kernel call, l - layer)

$$\textit{(FProp)} \quad \mathrm{FO}_{c,l} = \begin{cases} 0, c = 0 \text{ or layer inactive} \\ (\mathrm{FO}_{c-1,l} + \mathrm{BurstSize} \times \mathrm{BurstMult}) \mod \mathrm{OutDims}^2, \mathrm{Conv/Pool} \\ (\mathrm{FO}_{c-1,l} + 1) \mod \frac{\mathrm{OutDims}}{\mathrm{BurstMult}^2}, \mathrm{Fcon} \end{cases} \tag{4.14}$$

$$\textit{(FProp)} \quad \mathrm{MC}_{c,l} = \begin{cases} 1, c = 0 \text{ or layer inactive} \\ \mathrm{MC}_{c-1,l} + 1, \mathrm{Conv/Pool} \\ \mathrm{MC}_{c-1,l} + 1 \times (\mathrm{FO}_{c,l} == 0), \mathrm{Fcon} \end{cases} \tag{4.15}$$

$$\textit{(BProp)} \quad \mathrm{FO}_{c,l} = \begin{cases} 0, c = 0 \text{ or layer inactive} \\ (\mathrm{FO}_{c-1,l} + \mathrm{BurstSize} \times \mathrm{BurstMult}) \mod \mathrm{InDims}^2, \mathrm{Conv/Pool} \\ (\mathrm{FO}_{c-1,l} + 1) \mod \frac{\mathrm{InDims}}{\mathrm{BurstMult}^2}, \mathrm{Fcon} \end{cases} \tag{4.16}$$

$$\textit{(BProp)} \quad \mathrm{MC}_{c,l} = \begin{cases} 1, c = 0 \text{ or layer inactive} \\ \mathrm{MC}_{c-1,l} + 1, \mathrm{Conv/Pool} \\ \mathrm{MC}_{c-1,l} + 1 \times (\mathrm{FO}_{c,l} == 0), \mathrm{Fcon} \end{cases} \tag{4.17}$$

The weight setup varies depending on the layer and is calculated before computation starts to avoid overhead. For convolutional layers, in forward propagation, there are two kernels with every channel loaded in the FMem for each run - the one linking the input to the current output channel and the next one. As the output is calculated, the current and next kernels are updated with relation to the current output channel. For backpropagation, the process is the same, with the exception that every kernel of two error channels is loaded instead. When it comes to fully connected layers, the weight array contains every value connecting the current inputs and outputs, updating whenever different values are required. For backpropagation the process is the same, with inputs and outputs switched.

Regarding computation, convolutional layers are the least efficient, because they require the entire input to be written in memory before computation can start. Assuming BurstMult is setup optimally, pooling and fully connected layers finish computation one call after the previous layer finishes, requiring only one extra kernel call per layer added.

4.2.9.3 Forward Propagation

Algorithm 7 describes forward propagation. Each block is loaded, followed by the respective input beeing written. Computation is then issued, for the required number of calls, with the output beeing read when computation is finished. This cycle is repeated until every block runs once, leaving only the softmax calculation which is done by the CPU.

Algorithm 7 Network DFE Forward Propagation

Require:
Input
Ensure:
Output

```
1: for i ← 1:NBlocks do
2:     LoadBlock(i);                                   ▷ Load current Block
3:     WriteToLMem(BlockInput);                        ▷ Write Input Data
4:     for j ← 1:NCalls[i] do                          ▷ Compute Block Output
5:         RunKernel(Params[i][j], Weights[i][j]);
6:     end for
7:     BlockOutput ← ReadFromLMem();                   ▷ Read Result
8: end for
9: return Softmax(BlockOutput);                        ▷ Run Softmax
```

4.2.9.4 Backward Propagation

Backpropagation works mostly the same way as forward propagation. Blocks are instead ran in backward order and there is one additional parameter in the kernel call to contain the weight updates, which is used by the CPU to update the network weights after each block is ran. In addition, the softmax calculation is not mentioned in algorithm 8, as it's done automatically when the error is calculated, which is used as the input to the last block.

Algorithm 8 Network DFE Backward Propagation

Require:
Error

```
1: for i ← NBlocks:1 do
2:     LoadBlock(i);                                   ▷ Load current Block
3:     WriteToLMem(BlockInput);                        ▷ Write Input Data
4:     WriteToLMem(FwdInputs[i]);          ▷ Write required data for layer computation
5:     for j ← 1:NCalls[i] do              ▷ Compute Block Output and Weight Updates
6:         RunKernel(Params[i][j], Weights[i][j], WUpdates[i][j]);
7:     end for
8:     BlockOutput ← ReadFromLMem();                   ▷ Read Result
9:     UpdateWeights(WUpdates[i]);                     ▷ Update Weights
10: end for
```

4.2.9.5 Training

Before training, as shown in equation 4.1, the *BatchSize* parameter is extremely important in overall system performance.

The maximum amount of memory required for one block forward propagation is estimated to never be more than 40MB, using the VGG-16 network as reference and considering an extreme

case. Similarly, one block backward propagation is also estimated to never use more than 40MB of LMem, because, although more data is necessary per layer, the increased number of streams in the computation modules means each block must contain less layers. Since Galava LMem can hold up to 12 GB of data, *BatchSize* can be set to a maximum of 300.

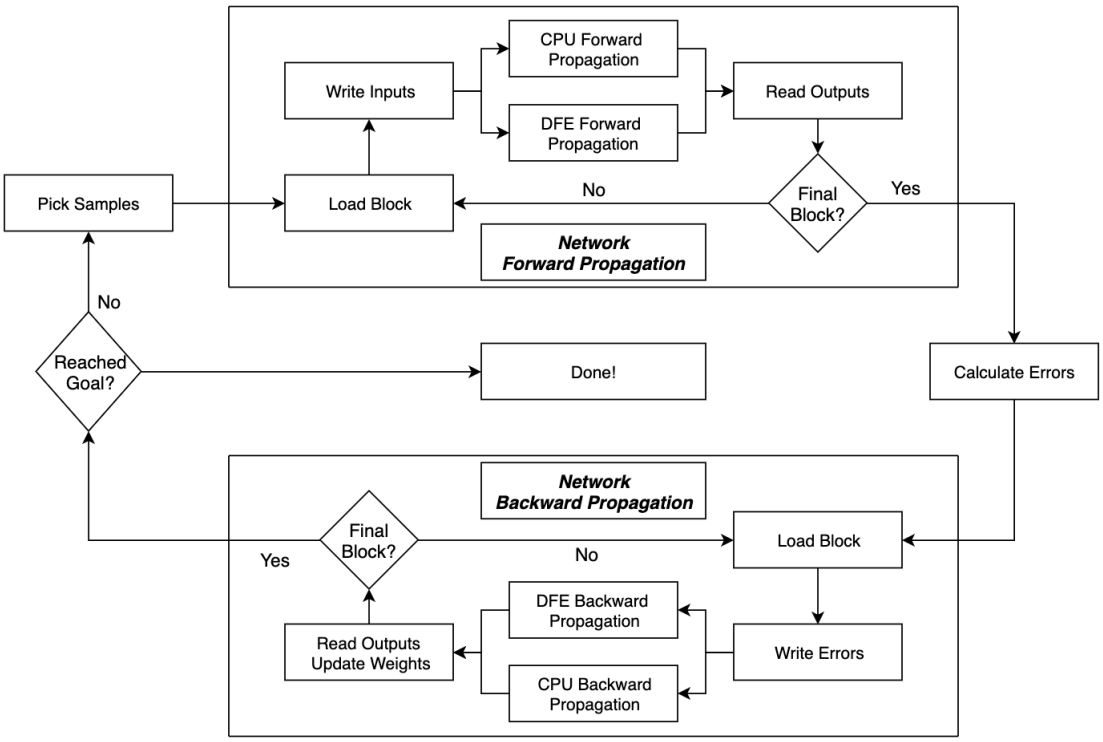


Figure 4.24: Training flow

Figure 4.24 describes the overall training flow. The chosen algorithm, as mentioned in chapter 3 is SGD.

In compliance with the algorithm, BatchSize samples are picked at random from the training dataset. Afterwards, the first block is loaded, inputs are written to memory and forward propagation starts. The CPU can be freed up while the DFE is computing by using nonblock functions to run the kernel, provided directly by the hardware compiler with no extra configuration. Naturally, because the DFE is faster than the CPU, only a few propagations can be performed while the DFE is running, which helps increase overall performance nonetheless. After each propagation completed by the CPU, the DFE completion status is checked. Finally, the outputs are read, saved and the next block is loaded. This process is repeated until all blocks have been used.

Continuing, the error is computed by the CPU using the chosen error function and the masks are calculated for pooling layer backpropagation. From here on, backpropagation starts is started, which follows what was said in section 4.2.9.4, with only one exception - weight updates are stored and each weight is only updated, with the average value, after every backpropagation is finished.

Once backpropagation is complete, the network is tested using a validation dataset. If goal accuracy is reached, training is complete and the model is saved for future use. Otherwise, the same process is repeated.

Chapter 5

Results

The design was tested using two popular CNN designs, AlexNet and VGG-16.

5.1 Validation

The first tests were done only with the C API. Each layer was tested separately at first, verifying correctness by comparing with the *tensorflow* python library. Afterwards, **AlexNet** and **VGG-16** were tested, with correctness once again beeing verified by comparing with *tensorflow*. *Valgrind* was also used, to make sure there were no memory problems with the code.

Once the previous tests were complete, the hardware code was devised. Starting with forward propagation, each layer was implemented separately at first and tested both in simulation, using the provided simulation tool, and hardware, by building the project to the DFE, with correctness was verified by comparing with the C API. In order to test an entire block, the manager was designed, with correctness beeing verified in the same way. Finally, forward propagation for entire networks was tested by running each block sequentially and validated using the same method as above. This entire process was repeated for backward propagation.

5.2 Network Configurations

The network configuration for both networks was done by using the C API.

Various values for both **Parallelism** and **Burst Mult** were tested, which are specified in section 5.3. Total execution time, as shown in equation 5.1, depends on layer execution time, $T_{Execute}$ and FPGA reconfiguration time, $T_{Reconfig}$ of each block.

$$\text{ExecutionTime} = \sum_{i=0}^{TotalBlocks} \text{TReconfig}_i + \text{TExecute}_i \tag{5.1}$$

$T_{Reconfig}$ has been previously discussed, in section 4.2.1.1, while $T_{Execute}$ varies on how large computationally intensive each block is.

Table 5.1: Network Configurations used for testing

Block	Validation Network	AlexNet	VGG
1	Conv, Conv, Pool	Conv, Pool, Conv	Conv, Conv, Pool
2	Fcon, Fcon	Pool, Conv, Conv	Conv, Conv, Pool
3	N.A	Conv, Pool, Fcon	Conv, Conv, Conv
4	N.A	Fcon, Fcon	Pool, Conv, Conv
5	N.A	N.A	Conv, Pool, Conv
6	N.A	N.A	Conv, Conv, Pool
7	N.A	N.A	Fcon, Fcon
8	N.A	N.A	Fcon

5.3 Resources

5.3.1 Forward Propagation

Synthesis was performed with very high effort and maxcompiler configured to optimize speed. Various frequencies were tested, but with frequencies higher than 150 MHz, it was significantly harder for the system to meet timing restrictions, regardless of effort, even with resource usage well below maximum.

Table 5.2: **AlexNet** DFE configuration and Resource usage

Block	BurstMult	Parallelism	Multipliers	Logic	FMem
1	125, 1000, 30	1, 1, 48	38.98%	76.91%	64.25%
2	1803, 7, 7	1, 64, 64	57.01%	79.30%	83.49%
3	5, 1, 6	48, 1, 24	53.40%	61.75%	86.14%
4	5, 4	12, 12	62.78%	53.63%	91.50%

Table 5.3: **VGG16** DFE configuration and resource usage

Block	BurstMult	Parallelism	Multipliers	Logic	FMem
1	2000, 2000, 2200	1, 32, 1	49.21%	67.64%	54.82%
2	520, 520, 555	64, 64, 1	48.82%	72.90%	95.53%
3	130, 130, 130	64, 32, 32	65.43%	81.83%	92.70%
4	33451, 33, 33	1, 64, 64	52.36%	68.23%	96.14%
5	33, 2000, 8	64, 1, 64	54.67%	64.87%	94.52%
6	8, 8, 2000	64, 64, 1	38.79%	63.34%	92.90%
7	5, 4	12, 12	36.10%	52.62%	90.23%
8	6	24	26.78%	34.56%	88.52%

5.3.2 Backward Propagation

Synthesis was performed with medium effort and maxcompiler configured to optimize speed.

Table 5.4: **AlexNet** 1 sample backward propagation.
F = 100 MHz, all times in *ms*.

Block	BurstMult	Parallelism	Multipliers	Logic	FMem
1	1, 1, 1	1, 1, 1	11.34%	30.93%	10.54%
2	1, 1, 1	1, 1, 1	8.61%	20.76%	13.67%
3	1, 1, 1	1, 1, 1	7.09%	14.23%	16.23%
4	1, 1	1, 1	4.56%	5.97%	14.03%

5.4 Performance

5.4.1 Forward Propagation

To prove design validity, the first comparison made was agaisnt an Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz CPU, using both AlexNet and VGG.

Table 5.5: **AlexNet** 1 sample forward propagation.
F = 150 MHz, all times in *ms*.

Block	CPU	This work	Speed-Up
1	2058.94	143.25	× 14.37
2	1727.20	347.97	× 4.96
3	1638.62	81.98	× 19.99
4	139.12	523.31	× 3.74
Total	5948.07	712.32	× 8.35

Table 5.6: **VGG16** 1 sample forward propagation.
F = 150 MHz, all times in *ms*.

Block	CPU	DFE	Speed-Up
1	12358.22	855.24	× 14.45
2	31934.66	1710.48	× 18.67
3	16373.57	1496.67	× 10.94
4	9099.75	997.78	× 9.12
5	12487.85	783.92	× 15.93
6	21810.05	926.51	× 23.54
7	1918.66	285.09	× 6.73
8	253.58	71.23	× 3.56
Total	106236.34	7126.92	× 14.91

For AlexNet, [3] only shows times for convolutional layer forward propagation, which were compared with the first three blocks of this work, including three pooling layers and one fully connected layer. Furthermore, only the DFE is used on both designs, meaning the nonblocking functionality is not used.

When not considering reconfiguration time, the achieved speed-up is $\times 0.02$. This mostly due to the extra layers computed, implementation differences and hardware differences. Naturally, computation in [3] is faster, because each layer has access every DFE resource available. Even so, the majority of the loss in performance can be attributed to hardware differences. [3] uses a station containing 8 DFE's, 38.4 GB/s LMem bandwidth and 8 GB/s PCI-E bandwidth, which is approximately $\times 38$ faster when compared the system used for this work, described in 4.1.1. However, considering not all of these factors are bottlenecks, a better estimation is around $\times 30$, which means the actual speed-up is actually closer to $\times \frac{10.72*30}{573.20} \approx 0.56$.

When reconfiguration time is considered, even with slower hardware, the designed system pulls ahead when it comes inference. Values for the reconfiguration times were discussed previously in section 4.2.1.1.

Table 5.7: **AlexNet** 1 sample forward propagation.
F = 150 MHz, all times in *ms*.

		F-CNN All convolutional Layers	This work First 3 blocks	Speed-Up
Computation		10.72	573.20	× 0.02
Total	Best (100ms)	510.72	873.20	× 0.59
	Average (500ms)	2510.72	2073.20	× 1.21
	Worst (1000ms)	5010.72	3573.20	× 1.40

5.4.2 Backward Propagation

[3] does not provide backpropagation results, so results were compared with the available CPU.

Table 5.8: **AlexNet** 1 sample backward propagation.
F = 100 MHz, all times in *ms*.

Block	CPU	DFE	Speed-Up
1	4325.17	567.27	× 7.62
2	3393.54	1407.09	× 2.41
3	1638.62	354.21	× 4.63
4	567.83	512.60	× 0.90
Total	9925.16	2841.17	× 3.49

Chapter 6

Conclusion

The presented hardware implementation of a CNN using the maxeler system outperforms [3], the only other CNN training system using maxeler, by, at best $\times 1.40$ and $\times 1.21$ on average, even with slower hardware. When it comes to training, while [3] can be faster for small networks, depending on the used batchsize, the proposed system is a much more scalable solution for deep networks, due to the reduced amount of reconfigurations required. Furthermore, the architecture is fully customizable and there is also a software implementation of a CNN which can run solely on CPU. The DFE has fully customizable parallelism by the user, which allows for different DFE models to be used, without any redesign effort beeing needed.

There are plenty of possible improvements, because the maxeler system has a broad range of capabilities, namely regarding system design. Since the system cannot be used for fast inference unless there is 1 DFE available per block, removing LMem padding and using the BatchSize as a means to make LMem streams align, as done in [3] might be a good way to improve performance, if only training is desired. Stream multiplexing might also be another possible improvement point, because the number of LMem streams is very limiting, expecially when it comes to backpropagation. Lastly, memory management can be improved, by splitting the kernel design into all the specific modules and using multiple kernels for each module, which could possibly allow for more parallelism.

Given the results, this work is current state of the art regarding convolutional neural networks using the maxeler system.

Appendix A

Bitwidth Configuration table

Table A.1: Allowable bitwidth configurations

Exponent Width	Maximum Mantissa Size
4	5
5	13
6	29
7	61
8-16	64

Appendix B

User Manual

B.1 Network Creation

```
// ------------------------ //
// ---- Network Struct ---- //
// ------------------------ //

typedef struct
{
  Block* Blocks;        // Network Blocks

  int TotalBlocks;      // Size of Blocks

  int BatchSize;        // How many inputs considered each time while training
  double LearningRate;    // How fast Network Learns
  double Momentum;      // How much previous changes to Weights influence current
    iteration
  char EFunc;         // Error function used to calculate error;

} Network;


// ----------------------- //
// ---- Init and Free ---- //
// ----------------------- //

  /*
    Initialize Network

    Net - Net to init
    InputDims - Dimensions of Input Volume for the Network

    return value - nothing
  */
  void InitCNN(Network* Net, int* InputDims);
```

```
32
33    /*
34      Free Network
35
36      Net - Network to be freed
37
38      return value - nothing
39    */
40    void FreeCNN(Network* Net);
41
42
43  // --------------------------- //
44  // ---- Blocks and Layers ---- //
45  // --------------------------- //
46
47    /*
48      Add a Block to the Network
49
50      Net - Network to add a block to
51
52      return value - nothing
53    */
54    void AddBlock(Network* Net);
55
56    /*
57      Adds a Conv Layer to the Current Block
58
59      NKernels - Amount of Kernels in this Layer
60      KernelSize - Size of Kernels in this Layer
61      Stride - Amount of Pixels Kernel Moves at a time
62      Padding - Amount of Pixels Added to Input Volume
63
64      return value - nothing
65    */
66    void AddConv(int NKernels, char KernelSize,
67           char Stride, char Padding);
68
69    /*
70      Adds a Pooling Layer to the Current Block
71
72      FilterSize - Size of Pooling Window
73      Type - Type of Pooling
74      Stride - Amount of Pixels Kernel Moves at a time
75
76      return value - nothing
77    */
78    void AddPool(char FilterSize, char Type, char Stride);
79
80    /*
```

```
    Adds an Fcon Layer to the Current Block

    Output Size - Amount of Output Neurons

    return value - nothing
  */
  void AddFcon(int OutputSize);

  /*
    Adds an Activation Layer to the Current Block

    Func - Activation function

    return value - nothing
  */
  void AddActi(char Func);

  /*
    Add a Dropout to the Current Block

    DropP - Dropout Probability

    return value - nothing
  */
  void AddDrop(double DropP);


// ------------------------- //
// ---- Edit Parameters ---- //
// ------------------------- //

  /*
    Set Batch Size

    Net - Network to consider
    Efunc - Batch Size

    return value - nothing
  */
  void SetBatchSize(Network* Net, int Bs);

  /*
    Set Learning Rate

    Net - Network to consider
    Efunc - Learning Rate

    return value - nothing
  */
```

```
130   void SetLearningRate(Network* Net, double Lr);
131
132   /*
133     Set Momentum
134
135     Net - Network to consider
136     Mom - Momentum
137
138     return value - nothing
139   */
140   void SetMomentum(Network* Net, double Mom);
141
142   /*
143     Set Error Func
144
145     Net - Network to consider
146     Efunc - Error Function
147
148     return value - nothing
149   */
150   void SetErrorFunc(Network* Net, char Func);
151
152   /*
153     Set Design Frequency
154
155     Freq - Frequency in MHz
156
157     return value - nothing
158   */
159   void SetDesignFreq(int Freq);
160
161
162 // ---------------------------- //
163 // ---- Pre-Defined Models ---- //
164 // ---------------------------- //
165
166   /*
167     Sets Network to AlexNet configuration
168
169     Net - Network that gets configured
170
171     return value - nothing
172   */
173   void CreateAlexNet(Network* Net);
174
175   /*
176     Sets Network to VGG16 configuration
177
178     Net - Network that gets configured
```

```
179
180     return value - nothing
181   */
182   void CreateVGG16(Network* Net);
```

Listing B.1: Network Creation Documentation

B.2 Inference and Training

```
1  // ------------------- //
2  // ---- Inference ---- //
3  // ------------------- //
4
5    /*
6      Classifies an Input with a given Network.
7      Only uses the CPU.
8
9      Net - Network to use
10     Input - Input to classify
11
12     return value - Classification Index
13   */
14   int ClassifyCPU(Network Net, double*** Input);
15
16   /*
17     Classifies an Input with a given Network
18     Only uses the DFE.
19
20     Net - Network to use
21     Input - Input to classify
22
23     return value - Classification Index
24   */
25   double* ClassifyDFE(Network Net, double*** input);
26
27
28 // ------------------- //
29 // ---- Training ---- //
30 // ------------------- //
31
32   /*
33     Train Network with a given Dataset.
34     Only uses the CPU.
35
36     Net - Network to be used
37     Inputs - Training DataSet. Dimensions need to be {DataSize, InDims}
38     Labels - Labels. Dimensions need to be {DataSize, NClasses}
39     DataSize - How many Inputs the Training DataSet contains
40     MaxEpochs - Maximum Amount of Epochs to run Training for
41     GoalError - Target Error
```

```
42    GoalAccuracy - Target Accuracy
43
44    return value - Nothing
45  */
46  void CNNTrainCPU(Network Net, double**** Inputs, double** Labels, int DataSize,
      int MaxEpochs, double GoalError, double GoalAccuracy);
47
48  /*
49    Train Network with a given Dataset.
50    Only uses the DFE.
51
52    Net - Network to be used
53    Inputs - Training DataSet. Dimensions need to be {DataSize, InDims}
54    Labels - Labels. Dimensions need to be {DataSize, NClasses}
55    DataSize - How many Inputs the Training DataSet contains
56    MaxEpochs - Maximum Amount of Epochs to run Training for
57    GoalError - Target Error
58    GoalAccuracy - Target Accuracy
59
60    return value - Nothing
61  */
62  void CNNTrainDFE(Network Net, double**** Inputs, double** Labels, int DataSize,
      int MaxEpochs, double GoalError, double GoalAccuracy);
```

Listing B.2: Network Usage Documentation

References

[1] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object Recognition with Gradient-Based Learning. In David A. Forsyth, Joseph L. Mundy, Vito di Gesú, and Roberto Cipolla, editors, *Shape, Contour and Grouping in Computer Vision*, Lecture Notes in Computer Science, pages 319–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[3] Wenlai Zhao, Haohuan Fu, W. Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-CNN: An FPGA-based framework for training Convolutional Neural Networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, July 2016.

[4] Neerja Doshi. Deep Learning Best Practices (1) — Weight Initialization, May 2019.

[5] J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462–466, September 1952.

[6] Rich Caruana, Steve Lawrence, and C. Lee Giles. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 402–408. MIT Press, 2001.

[7] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, September 2016. arXiv: 1609.04747.

[8] Peter Sadowski. Notes on Backpropagation, 2016.

[9] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv:1409.1556 [cs]*, September 2014. arXiv: 1409.1556.

[10] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *arXiv:1409.4842 [cs]*, September 2014. arXiv: 1409.4842.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[12] N. Voss, M. Bacis, O. Mencer, G. Gaydadjiev, and W. Luk. Convolutional Neural Networks on Dataflow Engines. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 435–438, November 2017.

[13] J. C. Ferreira and J. Fonseca. An FPGA implementation of a long short-term memory neural network. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, November 2016.

[14] Nelson H. F. Beebe. Accurate Hyperbolic Tangent Computation, April 1993.