FPGA IMPLEMENTATION OF REDUCED PRECISION CONVOLUTIONAL NEURAL NETWORKS

by

Muhammad Mohid Nabil

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Dinesh Bhatia, Chair

Dr. Poras T. Balsara

Dr. Mehrdad Nourani

To Mom, Dad and Tooba

FPGA IMPLEMENTATION OF REDUCED PRECISION CONVOLUTIONAL NEURAL NETWORKS

by

MUHAMMAD MOHID NABIL, BS

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2018

ACKNOWLEDGMENTS

This research is the culmination of a lot of hard work and support from a number of people. I would like to start by thanking Dr. Dinesh K. Bhatia for agreeing to be my supervisor and guiding me throughout the course of this research. I am grateful to him for not only being a teacher and advisor but also my mentor. Working at IDEA lab under his guidance has been the most fulfilling part of my academic journey at The University of Texas at Dallas. I would also like to thank him for meticulously going through my thesis, making valuable suggestions for improvement.

I am also thankful to Dr. Poras Balsara and Dr. Mehrdad Nourani for their guidance and being a part of my committee. I am thankful to Ritchie Zhao and M. Courbariaux for answering my queries on emails and helping to develop a better understanding of concepts during the early stages of my literature review.

I would like to thank all my lab mates for their words of encouragement during the course of this work. Their company made the long hours in the lab much more enjoyable.

In the end I would thank my entire family for their tireless support and prayers during the last 2.5 years. It would not have been possible without them by my side every step of the way.

June 2018

FPGA IMPLEMENTATION OF REDUCED PRECISION CONVOLUTIONAL NEURAL NETWORKS

Muhammad Mohid Nabil, MSEE
The University of Texas at Dallas, 2018

Supervising Professor: Dr. Dinesh Bhatia

With the improvement in processing systems, machine learning applications are finding widespread use in almost all sectors of technology. Image recognition is one application of machine learning which has become widely popular with various architectures and systems aimed at improving recognition performance. With classification accuracy now approaching saturation point, many researchers are now focusing on resource and energy efficiency. With the increased demand for learning applications in embedded devices, it is of paramount importance to optimize power and energy consumption to increase utility in these low power embedded systems.

In recent months, reduced precision neural networks have caught the attention of some researchers. Reduced data width deep nets offer the potential of saving valuable resources on hardware platforms. In turn, these hardware platforms such as Field Programmable Gate Arrays (FPGAs) offer the potential of a low power system with massive parallelism increasing throughput and performance.

In this research, we explore the implementations of a deep learning architecture on FPGA in the presence of resource and energy constraints. We study reduced precision neural networks and implement one such architecture as a proof of concept.

We focus on binarized convolutional neural network and its implementation on FPGAs. Binarized convolutional nets have displayed a classification accuracy of up to 88% with some smaller image sets such as CIFAR-10. This number is on the rise with some of the new architectures.

We study the tradeoff between architecture depth and its impact on accuracy to get a better understanding of the convolutional layers and their impact on the overall performance. This is done from a hardware perspective giving us better insight enabling better resource allocation on FPGA fabric.

Zynq ZCU-102 has been used for accelerator implementation. High level synthesis tool (Vivado HLS) from Xilinx is used for CNN definition on FPGA fabric.

TABLE OF CONTENTS

LIST OF TABLES

LIST FIGURES

CHAPTER 1
INTRODUCTION

1.1 Motivation

Image recognition tasks have gained immense popularity in the past few years. From the complex task of autonomous vehicles to simply unlocking a mobile device, recognition tasks are now considered essential to any embedded system. They also have other potential uses such as medical diagnosis and aerial vehicle control which require high levels of reliability in their performance. For all these applications, there is a need for high performance machine learning architecture which is highly accurate. Due to the mobile nature of a lot of these applications and their implementation on embedded platforms, they are also required to have excellent performance in terms of energy consumption.

In recent years, success with image recognition using convolutional neural networks (CNN) has made them exceedingly popular. CNNs became popular after the success of Alexnet in the 2012 ImageNet competition. This was the first time that CNNs had shown a promising accuracy. In the subsequent competitions, improved architectures of CNN were proposed each improving on the accuracy of the previously proposed scheme. In 2015, top-5 classification had achieved performance at par with human scores using GoogleNet with an error of 6.7%. These CNN architectures will be explained in later chapters for more clarity.

The basic building blocks of all these convolutional neural networks are the same. They consist of a layered architecture of convolutional, pooling, activation and fully connected layers. Together, multiple iterations of these layers form a dense structure of filters and neurons which work together to extract critical information for successful classification of input data. However, this dense architecture implies a large memory size along with a large computational complexity.

Graphical processing units (GPUs) have traditionally been used for neural network implementations. They are first used for the training phase and subsequently for inference (these concepts will be introduced in later sections). GPUs present an ideal platform for this process due to their highly parallel architectures. Extremely high throughput is achieved due to large number of processing elements (PE) available on chip. As evident, these systems have very high-power requirements which power these processing elements.

Field Programmable Gate Arrays (FPGA) present an ideal alternative to the above stated systems for CNN implementation. They provide a large number of programmable logic blocks which can be reconfigured on the fly. Massive parallelism can be achieved by efficient design on the FPGA

fabric while keeping power consumption at a minimum. This however, is not without its own set of challenges.

As explained above, deep networks have a large memory size and computational complexity. While FPGAs present an opportunity for high performance low power platform, their limited on-chip resources and memory limit the size of the convolutional neural networks. Owing to this, researchers are now focusing on low precision implementation for such platforms. This work addresses the problem of FPGA based CNN design for low precision networks.

Binarized neural network is one such low precision network which has recently been explored [1]. All data size is one-bit wide which limit the memory consumption as well as alter the way results are computed than traditional 16 or 32 bit precision networks on FPGAs.

1.2 Contribution of Thesis

This research has focused on various binarized network architectures and their implementation on FPGAs. In this regard, different structural combinations of the deep nets were studied to form a correlation between layers and prediction accuracy on a 32 bit precision network.

Due to the exploratory nature of this research work, we experiment with different machine learning libraries to get a better understanding of coding deep learning algorithms. In this regard, Tensorflow [2] and Caffe [3] were used for full precision network implementation. DIGITS from NVIDIA [4] is an excellent tool with Caffe backend which has been used for the results presented later in this thesis.

For binarized neural networks, we used a Theano based library from Courbariaux et al [5] which has been used widely by researchers working with low precision networks. Using this work, the trained weights are limited to -1 and 1 as per requirements of our binary model [6].

For hardware implementation of the network, zcu102 board from Xilinx has been used [7]. The board has over 32Mb of on chip memory which is sufficient to store trained neural network model. The large memory size was needed to eliminate off-chip memory access and reduce latency. We present a novel pipelined model which uses on chip memory access. This was important considering the high memory bandwidth requirements of the convolutional neural network layers

We use Vivado High Level Synthesis tool [8] for coding our design. This is an excellent tool which enables C++ level code to map on to hardware keeping a few Verilog constraints in mind. We will also explain a few of these limitations and guidelines in this report.

We also design the fully connected layer for binarized weights. Binary weights by virtue of their limited size eliminate the need for large multipliers greatly reducing resource and power consumption.

We also design a full precision convolution layer based on the well-known line buffer [9] for the first layer of our network as well as to benchmark differences between full precision and reduced precision layers.

It is important to note here that the training phase is always carried out offline using Graphical Processing Units and machine learning libraries and only the inference phase is mapped on to the FPGA fabric. This is because training is a one time or a periodic process and can be done offline. Inference is the more frequent process, and this is where lies the greatest potential for resource and power saving.

We will also explore the design space within the time constraints for this thesis. Different coding styles have different outcomes when it comes to hardware mapping and we will describe a few of these.

CHAPTER 2
BACKGROUND

We now introduce some important concepts necessary to understand the selection and implementation of binary weighted neural networks on FPGAs. We will also introduce some popular architectures of CNN, binarized networks and work from some previous researchers that influenced the work done in this research.

2.1 Network Components

We intend to explain the workings and principles of binary weighted convolutional networks. For this reason, we will start off from the very basics and try and explain the building blocks. In the following sections, we will explain the concept of perceptron. We will see how perceptrons join together into a web like structure to form an artificial neural network. We will then explain the need for a better structure and why we move towards convolutional neural networks. We will explain the additional components of the convolutional layers as well as give a brief account of some of the work done by researchers in this domain. We will look at binary weighted deep networks, component layers and each one of its benefits to us. We will also look at recent work by other researchers in this field.

2.1.1 Perceptron

Perceptron is inspired by a bio-computing model of a neuron in the brain. It falls in the category of linear classifiers. It works by proposing a hypothesis *h* which linearly separates any given data provided the data is linearly separable. The hypothesis divides the plane in two parts hence classifying data in 2 categories which has been illustrated in figure 1.

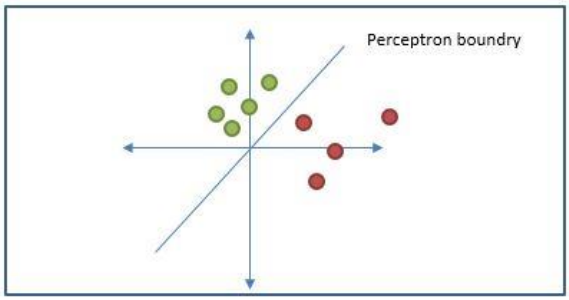


Figure 1: Illustration of linear decision boundry of a perceptron

For a two dimensional data, the perceptron works by adjusting weights for the two dimensions x1 and x2 through an iterative training process. The perceptron can be defined by the equation

$$w1.x1 + w2.x2 = \theta$$

We can see from the equation that we can adjust the separating line by adjusting parameters *w1*, *w2* and θ. From the above equation we can write our hypothesis as

$$g(x) = w1.x1 + w2.x2 - \theta$$

From this equation, we can get a simple linear classifier based on the sign of the left hand side. This "sign" activation is called the Activation Function or the Thresholding Function for the perceptron. This perceptron architecture including can be shown through the figure 2.

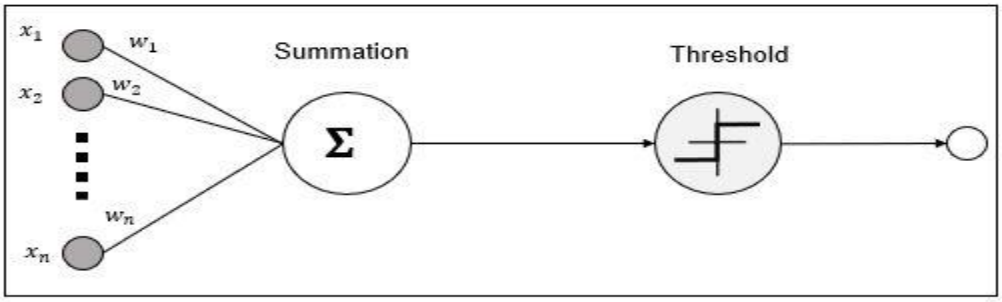


Figure 2: Perceptron Architecture

2.1.2 Perceptron Training Rule

As stated in the previous section, the perceptron equation draws a linear separation boundary using the *w1,w2* and θ parameters. These parameters are not arbitrarily decided. Rather, they are obtained through an iterative training process. The training process determines the parameters which would give the greatest separation between classes. Training data or labeled data is a dataset which is taken as the input to the training algorithm. Each input has a known output or a *label*. During the training process, this know label of input data is compared with the classifier output. The difference between the desired and actual output is used to update the network parameters. The way these parameters are updated is known as the training rule or the weight update rule.

We briefly explain the training rule for perceptron which updates the weights. This is because this rule forms the basis for the training of our eventual design of a convolutional neural network. Perceptron training (weight update) is done through a training data set using the following equations.

$$wi \leftarrow wi + \Delta wi$$

$$\Delta wi = n(t - o)xi$$

Where :

t = Target output of perceptron (also known as label of input data)

o = Actual perceptron output

n = step size constant

xi= training input

Δwi = weight update value

Due to the linear nature of the perceptron equation, there is an obvious limitation to classification when non-linearity is introduced in data. Since most real world problems have non linear data distribution, this calls for a better solution.

2.1.3 Neural Network

A neural network is formed by the connection of several neurons or perceptrons as illustrated in figure 3. Through interconnections of a large number of neurons, a web is formed which creates a highly non linear decision boundary for classification as illustrated in figure 4. The neural network is in the form of several layers. These are classified as input layer for the input side. Hidden layers for feature extraction. And output layer. There is one input layer, one output layer and many number of hidden layers as per need.

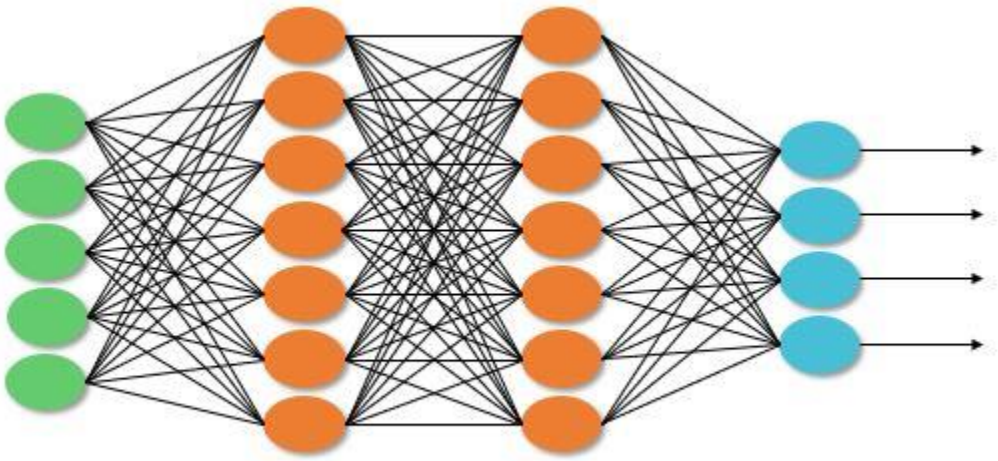


Figure 3: Neural Network Schematic inspired from [10]

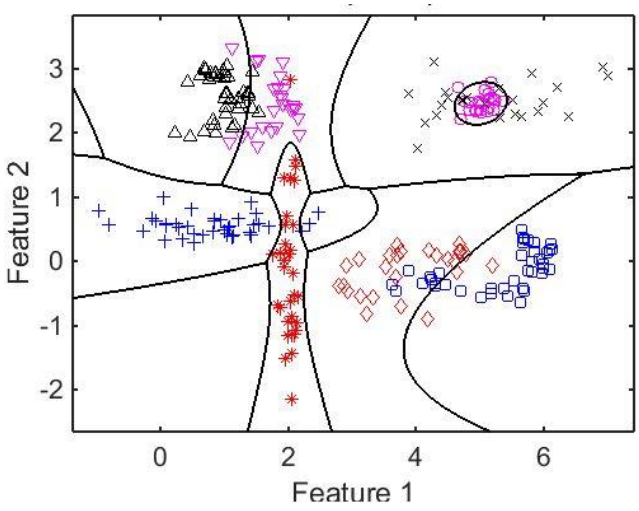


Figure 4: Illustration of a Non Linear Decision Boundary on Random Data Points

2.1.4 Neural Network Training

Similar to a perceptron, there is also a training rule for a network of perceptrons joined together to form a neural network. It achieves weight updates similar to a perceptron training rule. The algorithm to update weights based on a training data set is called the Backpropagation Method.

In this method, a set of data with labels is fed to the network. The outcome of the network is compared with the data labels and the error is determined. This error is then used to update the weights. These new weights are then used to repeat the process. This is called supervised learning process where the training data set has known output values which are used to train the network.

The error is determined in the first phase which is called the forward pass of data and the weight update is performed in the second phase known as the backward pass. Hence the name backpropagation.

We can summarize this algorithm as follows:

1) Initialize all weights (these can be any small number but not all zeros)
2) Until convergence, do the following steps
   DO:
   For each training example:
   DO:

a) Compute network output from the training input
b) For each output unit k, compute

$$\delta_k \leftarrow O_k(1 - O_k)(O_k - t_k)$$

c) For each hidden unit h, compute

$$\delta_h \leftarrow O_h(1 - O_h) \in w_{h,k}\delta_k$$

d) Update each network weight

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

Where

$$\Delta w_{i,j} = n\delta_{i,j}x_{i,j}$$

Steps a, b and c comprise the forward pass of the algorithm while step d comprises the backward pass of the algorithm.

We limit our discussion regarding the training at this point is since neural network training will not be a part of our research, and we will only be using the existing schemes.

We will however, include a brief note on the inference process by the neural network. As explained above, the network is arranged in the form of layers. Input data is fed in the input layer while the output is retrieved through the output layer. The hidden layer in between uses weight values to extract necessary features for classification. All these layers work together to get the desired result.

What we have not explained yet is what goes into the input layer. Consider an image for classification. What part of image are we supposed to feed in to the network to classify it. These inputs or features are acted upon by the network weights for classification. An in depth knowledge of the area of classification is necessary to determine which features would help in classification.

To take another example, for speech classification tasks, an area expert would know that formant frequencies *f0* and *f1* (speech features) are required for some speech recognition tasks. This domain specific knowledge greatly restricts the use of neural network as an effective classification algorithm. Also, for data with very large feature size such as a 2D or a 3D image, this is a grave limitation. A more generic solution to this is hence required.

2.1.5 Convolutional Neural Network

In recent years, Convolutional Neural Networks have become increasingly popular for image classification tasks due to their exceptional ability to handle very high input feature size as in the case of image recognition task. They have also been widely used in other applications such as machine vision and data centers[11-13].

CNNs have a layered architecture with different layers performing different tasks. The layers are then repeated multiple times to extract more and more information from the data as well as to reduce the data size. These layers and their functions will be explained in more detail in the subsequent sections.

The input to each layer in the CNN is called a feature. All the features to one layer are collectively called feature map. Each layer has its own set of input or feature map which have varying dimensions. Except for the input feature, which is the input to the first layer, all subsequent feature map dimensions depend upon the layer architecture producing these features. Hence the output feature map of one layer is the input feature map of the next.

2.1.6 Working Principle

As explained above, convolutional neural networks are ideal for data with very large feature size. Image classification is one such example of this. Consider an RGB image with dimensions 32x32 (height x width). With three layers for red, green and blue, we get a total number of 3072 input features. A convolutional neural network needs to be able to tackle two problems associated with this data. First is to reduce the number of input features into a more manageable number. This is achieved through pooling layers in the convolutional neural networks that reduce the number of features in each subsequent layer of the layered architecture. Second is linked to the first, which requires the extraction of only relevant features for successful classification of this image. This extraction is done by the convolutional layers of the neural network which is achieved through training the network weights. These weights are convolved with the features to extract requisite information i.e. most relevant data to successfully classify the image. Both these layers and others in the architecture will be explained in the next section for more details.

2.1.7 Architecture

As explained above, CNN are made of a layered architecture. These layers take input feature map and transform it into an output feature map. This is then processed by the next layer in the architecture.

2.1.8 Convolutional Layer

Consider an input image of dimension $im_{height}\ x\ im_{width}$ which is fed to the convolutional layer. This layer is formed up from a number of filters which are also referred to as kernels in the literature.

Each filter has dimension $f_{height}\ x\ f_{width}$. Suppose we have one filter in the layer. This filter is convolved over the input image. At each point of convolution, the filter returns one pixel or value as per the method of convolution. Hence for each input pixel, we get an output and the total output feature map can be extracted by the following equations

$$Out_{x,y} = \sum_{i=1}^{f_{height}} \sum_{j=1}^{f_{width}} im_{x+i,y+j} \; x \; filter_{i,j}$$

Where *x,y* represent the pixel position in the output map which is obtained by a 2 dimensional multiply accumulate operation between the input image and filter

Similarly, for high dimensional input feature map, we have a total number of N input feature maps which are reduced to a single layer at the output by one filter of dimension *height x width x N* and is given by:

$$Out_{x,y} = \sum_{k=1}^{N} \sum_{i=1}^{f_{height}} \sum_{j=1}^{f_{width}} im_{k,x+i,y+j} \; x \; filter_{k,i,j}$$

Figure 5 illustrates this multiply accumulate operation. We notice here that the output for one filter is always expressed in a 2 dimensional plane represented by *x* and *y*.

Commonly used filter dimensions are 3x3 and 5x5. These have been obtained through empirical testing and have been widely adopted. An important point to note here is that due to the nature of 2D convolution that has been done here, the output dimension is always less than the input dimension. For a given input of dimensions *Input Dim x Input Dim*, we get an output of dimension *Output dim x Output dim* where each *Output dim* is given by:

$$Output\ dim = Input\ Dim - Filter\ Dim + 1$$

For example, for a symmetrical input of size 32 x 32 and filter size of 5 x 5, the output would be reduced to 28 x 28. To keep the architecture symmetric, the output is usually *zero padded* to make it 32 x 32 again. Zero padding is a process where the size of the feature map is increased by adding zeros to it. This is done on the periphery of the feature map i.e along the outer edges. We will see how this seemingly troublesome property would be used to our advantage in subsequent layers.

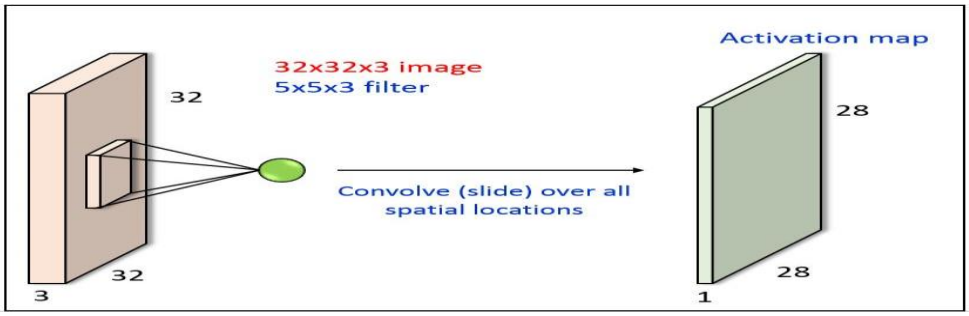


Figure 5: Convolution Layer Operation

2.1.9 Activation Layer

Similar to what we discussed in perceptron and neural network, activation function is also applied in convolutional neural networks. The layer takes convolutional layer output maps as input and applies an activation function to it. For example, if the activation function was a simple sign function, we would get

$$Output_{x,y} = sign\ (Input_{x,y})$$

There are several other activation functions used in CNN. A very commonly used function is the Rectified linear unit (RELU) which can be shown by equation below

$$Output_{x,y} = max\ (Input_{x,y}\ , 0)$$

Other commonly used activation functions include:

Sigmoid:

$$Output_{x,y} = \frac{1}{1 + \ e^{-Input_{x,y}}}$$

PReLU:

$$Output_{x,y} = constant \text{ x } Input_{x,y}\ for\ Input_{x,y} < 0$$

$$Output_{x,y} = Input_{x,y}\ for\ Input_{x,y} \geq 0$$

ELU:

$$Output_{x,y} = constant\ (e^{Input_{x,y}} - 1)\ for\ Input_{x,y} < 0$$

$$Output_{x,y} = Input_{x,y}\ for\ Input_{x,y} \geq 0$$

These have been shown in figure 6.

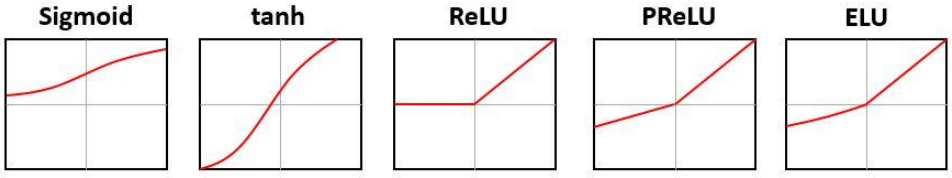


Figure 6: Different Activation Layer Curves

2.1.10 Pooling Layer

As described in previous section, one of the consequences of convolving a filter over an image is the reduction in size. This corresponds to a reduction in the number of pixels. Smaller image size means smaller computation and memory footprint during the multiply accumulate operation. We use this property in pooling layer. There are several ways we can do pooling. One popular method is called the max-pool. In this method, the filters are moved over the 2D image. Instead of a MAC operation as is the case in convolution, we take the max value of image pixels overlapped by the filter at a given instant. We can also do an average operation instead of the max operation as is done in average-pooling.

We can control the extent of the size reduction by controlling the stride s of the filter over the image. A stride is the step size the filter takes while convolving over the image. For convolutional layer, the stride s=1 is a commonly used value. We can extend the output dimension equation with variable s in it as follows

$$Output\ dim = (Input\ Dim - Filter\ Dim)/s + 1$$

Through observation of the above equation, we can see that increasing stride would reduce the Output dim.

2.1.11 Normalization

Normalization layer has recently gained popularity and is increasingly being used for training of neural networks. It has been empirically shown that training convergence is achieved much faster with applying a normalization factor to batch. Training process requires careful selection of initial weight values. Otherwise, the training process becomes exceedingly slow. Batch normalization shifts data to zero mean and unit variance. This drastically reduces training times and resource utilization [14] and solves the convergence issue [15].

We will explain this more in the binarized network section since it is more commonly used in reduced precision networks.

2.1.12 Fully Connected Layer

A fully connected (FC) layer is the artificial neural network explained in the previous sections. This layer has usually no hidden layer since feature extraction is not really required which is done by the convolutional layers. More than one fully connected layer is usually a part of the CNN architecture. Since no convolution is performed in this step, data is more conveniently expressed as an array rather than an image prior to feeding it to the fully connected layer. This has been illustrated in figure 7.

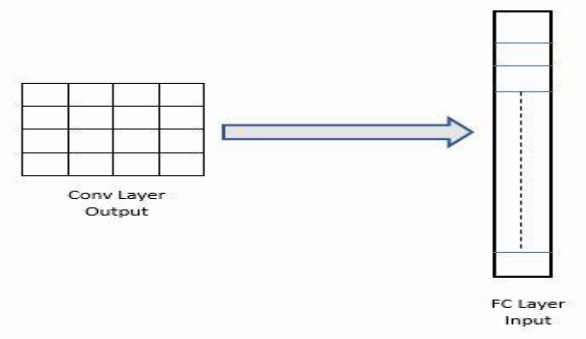


Figure 7: Conversion of data representation from image to an array

2.1.13 Softmax Layer

Softmax layer is used for classification which converts raw numerical values into probabilities of classes. This is the final layer of the CNN architecture. Each output node from this layer represents probability of the class value. Sum total of all outputs is equal to one. Probability of each class is given with the below equation which is implemented in the softmax layer.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Where x_j represents raw output from the final FC layer for each class *j*. $\sigma(x_j)$ represents probability of class *j*. $\sum_i e^{x_i}$ is summation for all classes *i* from 1 to *j*.

2.2 Popular Datasets

We will now briefly describe some image data sets which will be repeatedly referred to in subsequent sections. These data sets are commonly used test cases to test neural network performance.

2.2.1 MNIST

MNIST [16] is a set of labeled images of handwritten digits from 0 to 9. It has 60,000 training samples and 10,000 test samples. The image size is 28 x 28 pixels for each of the samples. The data set has been widely used to test smaller CNN architectures such as LeNet [17]. Figure 8 shows some sample images of MNIST data set.

Figure 8: MNIST Dataset Images

2.2.2 CIFAR-10

CIFAR-10 [18] is another popular dataset. The postscript 10 represents the total types of classes it has. CIFAR-10 has 60,000 RGB images which are divided into 10 classes. Each class has 6000 images. The total number of images is divided into 50,000 training samples and 10,000 test samples. Each image is of size 32 x 32 x 3 for the RGB layers. The images have the following classes; airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Figure 9 shows sample images and their classes from CIFAR-10.

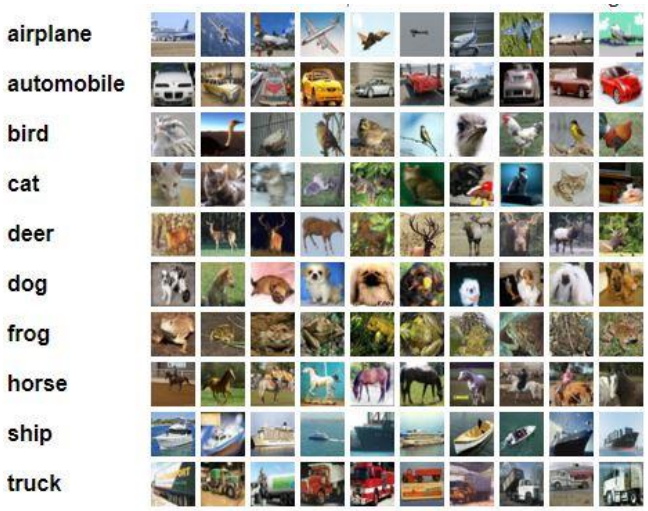


Figure 9: CIFAR-10 Dataset Images [18]

2.2.3 ImageNet

Imagenet [19] database is the part of the Imagenet Largescale Visual Recognition Challenge (ILSVRC) which is held annually where designers compete to achieve the highest classification accuracy on the an immense data set of 14 million images divided into a 1000 categories. It is the most comprehensive set of images and has been used to gauge the performance of the best machine learning algorithms. The samples consist of 256 x 256 RGB images. The dataset is a result of a

massive crowdsourcing effort carried out by researchers from Princeton. The images however, are not owned by the group and it only provides URL to each image which can be downloaded as a set. More details can be obtained from [19].

2.3 CNN Architectures

We now give a brief description of some architectures which have outperformed other learning algorithms in the ImageNet challenge and hence made CNN the go to architectures for many machine learning applications.

2.3.1 AlexNet

AlexNet [20] was the first CNN architecture to win the ImageNet challenge in 2012. This architecture proposed by Krizhevsky et al. Alexnet exploits the parallel computation capability of traditional graphical processing units (GPU) and divides each leyer in the network between two GPUs.

AlexNet consists of a total of 8 layers. Of these, 5 are convolutional layer and 3 are fully connected. After the last layer, there is a softmax layer. After each convolutional layer, there is a ReLU activation layer. There are two normalization layers after the first and the second convolutional layers. There are three max pool layers after first, second and fifth convolutional layers. These follow after the normalization layers. Their model achieved a top-5 error rate (make 5 class predictions for each image) of 17.0%. Figure 10 has been adopted from the original work and illustrates the architecture.

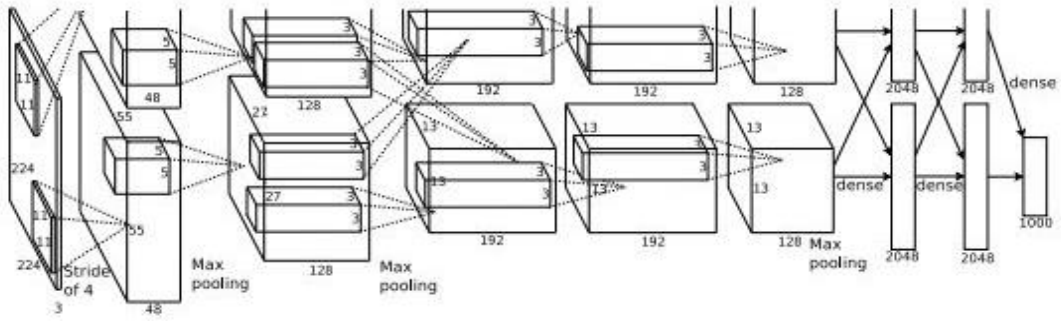


Figure 10: AlexNet Architecture from Authors Original Work [20]

2.3.2 VGG

VGG [21] is a very deep convolutional neural network exploiting extremely high computational power of modern GPUs. It has 16 and 19 layers for VGG-16 and VGG-19 respectively. Furthermore, it has 5 max pool layers and a softmax layer at the end of the architecture. Figure 11

shows the configuration extracted from the original paper [21]. It was the winner of Imagenet challenge in 2014. It has a very symmetric architecture with all filers of size 3 x 3 which make it convenient to implement. The extremely deep architecture corresponds to very high computational requirements with over a 100 million training weights and around 16 billion MAC operations. Their most dense model gave a top-5 error of 7.5% which was substantially better than the previous models.

In figure 11, we can see various configurations tested by the authors. For the most dense architecture of VGG-19 (column E), we can infer the following network configuration. It has a total of 19 weight layers. It takes a 224x224 RGB image as an input. First and second convolution layers have 64 filters each. After two convolution layers, there is a pooling layer. This is followed by two more convolutional layers with 128 filters each. After this there is a pooling layer. This is then followed by 4 convolutional layers with 256 filters each and then a pooling layer. In the next 4 layers there are 512 filters each followed by a pooling layer. Finally, 4 more convolutional layers with 512 filters each is added which is again followed by a pooling layer. After all the convolutional steps are done, 3 FC layers are added. Hence the total number of convolutional and fully connected layers is equal to 19.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 11: VGG Layer Configurations [21]

2.3.3 GoogleNet

GoogleNet [22] was a revolutionary architecture presented to overcome the drawbacks of VGG i.e. high resource and computational requirement. It gave a top-5 error rate of only 6.7% while only requiring 1.2 million weight values. In addition, it drastically reduced the computation requirements with only 800 million multiply accumulate operations making it suitable for use with low power embedded devices. The architecture achieves this by forming a network in network architecture, small sections called the inception modules. Without going in to too much detail, we conclude that this presented a more feasible solution than the very deep VGG.

2.3.4 SqueezNet

There have been models with an even better performance than GoogleNet such as ResNet. However, they rely on extremely deep architectures with high resource utilization. Acting on a different ideology of reasonable precision with reduced size, squeezenet [23] aims to achieve performance similar to AlexNet with drastically reduced resource consumption. This model presented by researchers from UC Berkley was created specifically for embedded domain making them even feasible for FPGA implementation. Several researchers have been working in this field with very promising results. The architecture relies on ‘Fire Modules’ to squeeze and expand the layers using 1x1 and 3x3 filters. We will not go into the detail of these modules.

CHAPTER 3

CNN ANALYSIS AND BINARY WEIGHTED NETWORKS

3.1 Introduction

Since the main goal of this research is the implementation of CNNs on FPGA, it is pertinent to analyze CNN architectures with respect to their resource utilization. This includes both the memory requirements as well as the MAC operations which constitute the major portion of power consumption on a chip. This section is important since we will start to build our case for the use of binarized neural networks over the traditionally used full precision CNN.

3.2 Memory Analysis

Consider the example of VGG-16 which is a very deep net with 16 layers comprising of convolution as well as fully connected layers. The total network has over 130 million weight values. If we were to have 16 bit values for each weight, this would translate into 260MB of memory. Most of these weight values are concentrated in the fully connected layers of the network. It was documented in [24] and figure 12 shows the distribution of weights in various CNN layers of VGG 16 and VGG 19 architecture.

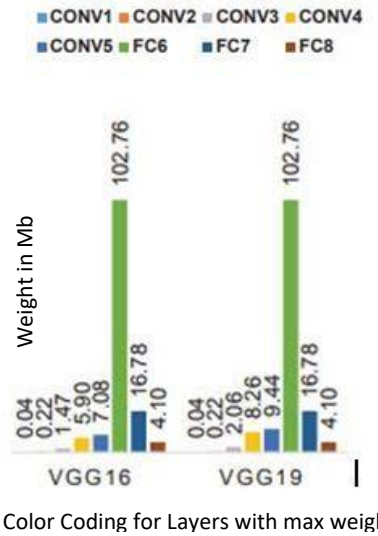


Figure 12: Distribution of weights in VGG 16 and VGG 19 from [24].

As evident from figure 12, the weight distribution of the network is such that the weights are overwhelmingly concentrated in the fully connected layers. Addition of more FC layers in any architecture would hence drastically increase memory consumption. For systems such as GPUs

with very high available memory resources, the network is well suited for implementation. This weight distribution can also be shown in figure 13 adopted from [1] which elaborates the marked difference in memory utilization between conv layers and fully connected layers.

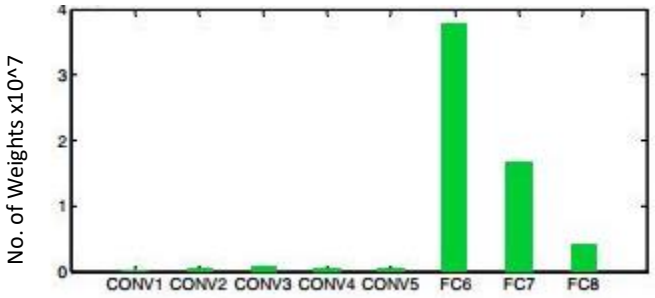


Figure 13: Memory analysis of AlexNet from [1].

To better understand the memory consumption of weight values, we perform a little analysis to see the relation between increasing weights and memory utilization.

Suppose we have a fully connected neural network layer. We have 2 neurons in the input and 1 neuron in the output. We have a total of 2 weight values. Increasing neurons in the input layer by one would add one more weight to our memory bank. Now we take a network with 2 output neurons. Increasing input layer neurons by a factor of x would increase our weight values by a factor of 2 . We can form a more general relationship between our input and output neuron weights as:

$$Increase\ in\ Weights = \ Increase\ in\ Input\ Neurons\ x\ Output\ Neurons$$

From above relation, we can easily see that the storage elements explode with even a fairly small number of neuron values. For large data sets such as Imagenet with a large number of output classes, even one fully connected layer would have a multiplying factor of a 1000 and hence an exceedingly large number of weights.

Now let us consider the hardware storage requirements of such a network. Each storage element or weight in the network has an associated precision with it. This depends on the number of bits allocated for the storage element. Considering a 32 bit floating point storage requirement for each weight, we can modify the above equation to get bytes of memory

$$Increase\ in\ Weights(Bits) = \ Increase\ in\ Input\ Neurons\ x\ Output\ Neurons\ x\ 32$$

We take the example of VGG and see how the memory is distributed in each layer in table 1.

Table 1: Resource Analysis of VGG Network

Input =224x224x3				
Layer	Inputs	Filters	Weights	Memory(32 bits/weight)
Conv1	224x224x3	64	1728	55.2Kb
Conv2	224x224x64	64	36864	1.1Mb
Maxpool				
Conv3	112x112x64	128	73728	2.3Mb
Conv4	112x112x128	128	147456	4.6Mb
Maxpool				
Conv5	56x56x128	256	294912	9.4Mb
Conv6	56x56x256	256	589824	18.8Mb
Conv7	56x56x256	256	589824	18.8Mb
Maxpool				
Conv8	28x28x256	512	1179648	37.7Mb
Conv9	28x28x512	512	2359296	75.4Mb
Conv10	28x28x512	512	2359296	75.4Mb
Maxpool				
Conv11	14x14x512	512	2359296	75.4Mb
Conv12	14x14x512	512	2359296	75.4Mb
Conv13	14x14x512	512	2359296	75.4Mb
Maxpool				
FC1	8192	4096	33554432	1073Mb
FC2	4096	4096	16777216	536Mb
FC3	4096	1000	4096000	131Mb
Softmax				

From table 1, we see the immense storage requirement of a deep network such as VGG-16. We have considered each weigh value to be a full precision 32 bit storage which would give us the best performance.

At this point, we will conclude our discussion on deep networks and their memory requirements. We move on to the processing requirements of such networks.

3.3 Processing Requirements

Neural network is nothing but an exceptionally large number of multiply accumulate operations on data to separate them as per a highly non-linear boundry. As explained in previous sections, convolutional and fully connected layers work in coherence to first extract the most relevant features from the data and then separate them on the basis of those features.

We have seen in section 2.1.8 that convolution is a MAC operation performed in 2 dimensions in its most simplest form on a single layer of input. Increasing the number of input layers to any given

layer increases the number of MAC operations by a factor of the depth of the input layer. To get a realization of the number of operations required, we take an example of a 4x4 image convolved with a 2x2 filter to get a 3x3 output image. At one instant of convolution, we have a total of 4 multiplies and 4 accumulate operations. We have a total of 3 parallel strides and 3 horizontal strides to get a 3x3 output. Hence we get a total of 3x3x4 multiplies and 3x3x4 accumulates. This is a very large number considering the small size of our filter and input image.

We take another example of a fully connected network of 4 inputs and 3 outputs. We get a total of 4x3 multiplies and 4x3 accumulates. We will see that as we grow the network bigger, this number does not grow as drastically as the convolutional layers. This was again documented in [1] from where figure 14 has been adopted which depicts the distribution of MACs in a 5 convolution 3 FC deep network.

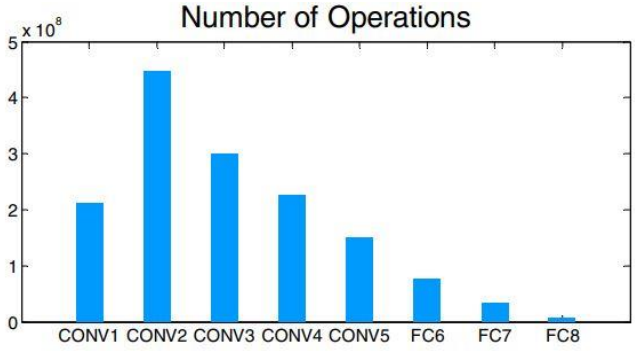


Figure 14: Processing Requirements of CNN Layers of AlexNet [1].

We will conclude this section with the result that in any deep network, convolutional layers comprise the greater portion of operational load on the system.

3.4 Reduced Precision Networks

As shown in previous sections, convolutional neural networks have a very large memory and resource requirement. In order to optimize any implementation, the algorithm has to cater for the high processing need at convolutional stage and high memory consumption at the fully connected stage. One way to take care of these two problems is to reduce the bit width of storage type.

A growing number of researchers are looking at this approach through capitalizing on the inherent redundancy of deep networks. Researchers have shown that reduction in data width does not adversely impact network accuracy. Han et el. [25] demonstrate the use of efficient optimizations such as compression paired with reduced bit width in reducing the architecture size by more than 40 times. This is a critical advantage necessary for neural network implementations in memory

constrained architectures such as FPGAs. Other researchers [24, 26, 27] have argued on similar line showing minimal loss of accuracy with bit width drastically less than 32 bits.

There have been various implementations of neural networks with reduced bit widths. These include 16 bit, 8 bit , 2 bit as well as 1 bit implementations. The choice of data width has significant impact on the hardware architecture the designer selects. We focus on binarized implementation of the deep network and we will explain in this section the architectural differences in a binarized network and how it tackles the earlier mentioned problems of full precision convolutional neural networks.

3.5 Binary Convolutional Neural Networks

Binarized neural networks with binary weights and activations have recently shown impressive performance almost close to a full precision network. Binary networks are somewhat similar to their full precision counterparts in terms of layer arrangement. The two networks have been contrasted in figure 15 which as been adapted from [1]. Each layer of the binarized network as shown in figure 15 will be explained in the subsequent sections.

In a binarized neural network, all weight values as well as input maps have values restricted to either 1 or a -1. In a hardware system, we can use a 1 bit data type to represent these values. A 1 would represent 1 and a 0 would represent a -1. All arithmetic operations are performed using these one bit data types. They also determine the storage requirements of input maps and weights in memory.We now contrast these layers with the full precision network and see how we get these savings.

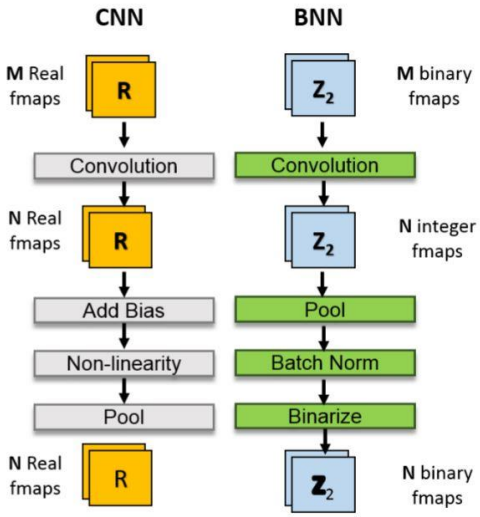


Figure 15: Comparison of CNN and Binary Networks adapted from [1].

3.5.1 Binary Convolution

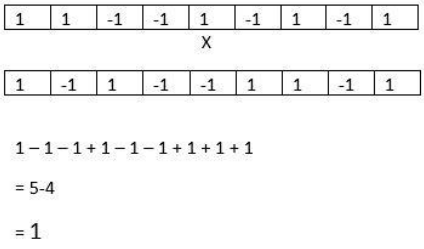
We stated before that most of the processing in a deep network is done in the convolutional layers. A 32 bit multiplier can be implemented in a number of ways depending upon the underlying hardware. On an FPGA, this operation can be performed using the limited on chip DSPs. They can also be implemented using LUTs and FFs. We design a multiplier using Vivado HLS tool and see the resource utilization for fully unrolled multiplication of two 3x3 array of 32 bits in table 2.

Table 2: Binary Convolution Resource Requirement

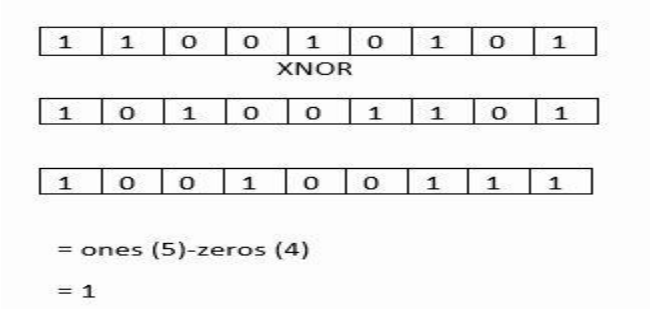
DSP	FF	LUT
03	15	80

These are fairly large numbers since they have to model a series of full and half adders. Convolution whether in a full precision network or a binary network is nothing but a repeated process of matrix multiplication and summation. If we were to add the addition overhead to the above figures they would no doubt be even greater.

We now consider the convolution process in a binary network. In this network both the input layer as well as the feature matrix has a binary data.



We can see that this is nothing but an XNOR followed by a bit counting operation between two arrays. Bit count is also sometimes referred to as popcount. In this step, we count the number of ones and subtract the number of zeros from this count. This is further elaborated below:



3.5.2 Batch Normalization

Batch normalization is used to reduce the training time by faster convergence. As explained in previous sections, training set is used to update weights by back propagation for neural network training. However, the choice of initial weights is critical in this process since it determines convergence time. Batch normalization [14] is a process which eliminates this problem by statistical mean and variance shift using batch normalization parameters calculated at training time. The transform equation is given below:

$$output\ pixel = \frac{input\ pixel - u}{\sqrt{\sigma^2 + \varepsilon}}\gamma + \beta$$

$u, \sigma^2\ and\ \varepsilon$ are the mean, variance and *hyper parameter* respectively. Mean and variance are statistical parameters calculated from each batch. A batch is a subset of the given training set after which the network weights are updated. *Hyper Parameter* is a constant set to adjust training time. $\gamma\ and\ \beta$ are parameters obtained during the training process. The statistical parameters shown in equation above are applied to the output data after each batch is processed, hence the name batch normalization. The training method is given in [14]. Each input feature map has its own normalization parameters. This expression can be simplified to other forms offline before hardware inference stage to make processing easier. Batch normalization has found application in reduced precision networks more than full precision since convergence problem is much more amplified with reduced precision due to loss of information. This can be seen in the figure 16 from [15]. We see that for a binarized network (figure 16 right), CNN with batch normalization converges much faster than CNN without batch normalization with increasing number of epochs. An epoch is one complete cycle through the entire training data set. We see that even after 200 cycles through the training data (epochs), network without normalization has a high error. This phenomenon is also present in a floating point CNN (figure 16 left), however, it is not as pronounced.

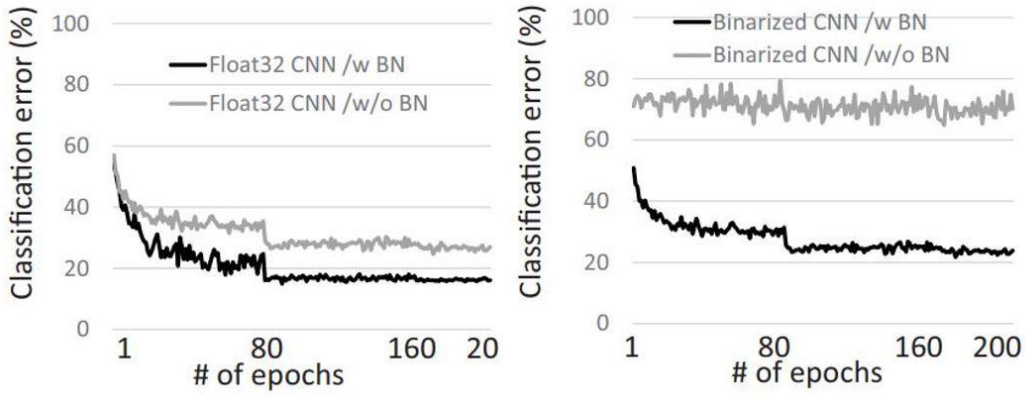


Figure 16: Impact of Normalization Layer on Convergence During Training

3.5.3 Binarization Layer

Binarization layer is a simple activation function which reduces the bit width back to single bit precision at the end of the convolution process. This can be expressed by a simple activation function such as:

$$output = 0 \leftarrow sign(input) < 0 \; and \; 1 \leftarrow sign(input) \geq 0$$

3.5.4 Fully Connected Binary Layer

In binarized networks, fully connected layers are also binarized. Each weight value is one bit which is multiplied (XNOR-popcount) with a 1 bit input vector. In the fully connected layer, we get integer values in between the input and output of the layer which needs to be binarized as shown above. Also due to reasons explained in the batch normalization section, reduced precision FC layers also need normalization for better performance. We can expand the FC layer to see data widths as shown in figure 17.

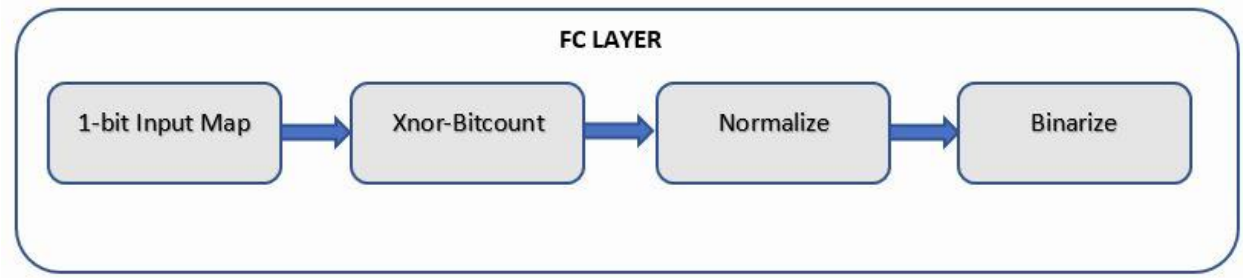


Figure 17: Fully Connected Layer Operation

Now that we have explained all the fundamental concepts necessary for convolutional neural network implementation, we will describe the training phase and network analysis.

CHAPTER 4
NETWORK TRAINING

4.1 CNN Training Setup

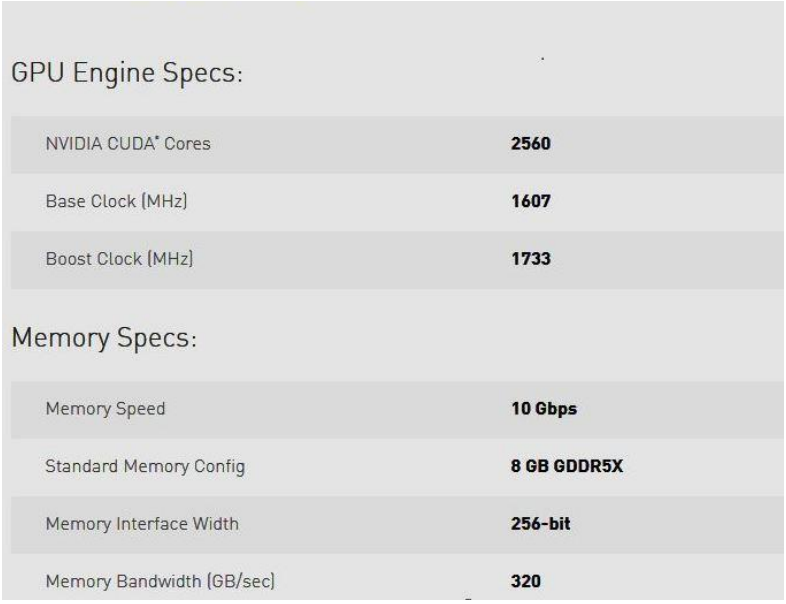
The first step for neural network training was to create the appropriate workstation for the task. In the following sections, we will discuss the hardware platform required, various training libraries and systems tried out for different types of networks, detailed analysis of network and its impact on classification of CIFAR-10 dataset and reduced precision training set up used.

4.1.1 PC Based

We started the process by first implementing a small cnn model (LeNet) [17] for MNIST dataset. This was done using Tensorflow from google. Tensorflow is a very powerful machine learning library which can be used with a Keras [2] front end for easy implementation of CNN architectures. The entire setup included: Anaconda [28] was installed which is an excellent tool to handle python libraries and environments. A separate environment for different types of networks was created along with required python version. Tensorflow with Keras front end was installed and tested on MNIST. The network achieves accuracy of over 99% with this small dataset. One thing that quickly became clear was the exceedingly large amount of time required for training on a CPU based system. For this reason we moved to a larger GPU based workstation for further testing.

4.1.2 GPU Based Workstation

GEFORCE GTX 1080 GPU from NVIDIA was subsequently used for the remainder of training sessions. The workstation consisted of Intel Xeon processor with two NVIDIA GPUs. Each GPU has 8GB of RAM available on board and a large number of processing cores for extremely high throughput. The workstation had two PCI slots enabling simultaneous use of two GPUs by using CUDA aware MPI [28]. However, We decided to use the two GPUs independently for training two separate networks in parallel as a large number of networks had to be trained to study impact of parameter tweaking. Specifications for the GPUs used have been shown in figure 18.



GPU Engine Specs:

NVIDIA CUDA® Cores	**2560**
Base Clock (MHz)	**1607**
Boost Clock (MHz)	**1733**

Memory Specs:

Memory Speed	**10 Gbps**
Standard Memory Config	**8 GB GDDR5X**
Memory Interface Width	**256-bit**
Memory Bandwidth (GB/sec)	**320**

Figure 18: Specifications of GPU Used for Network Training [29].

4.1.3 Software Setup

As part of the research, we wanted to study the impact of different network parameters on classification accuracy (this will be explained in more detail in subsequent section). This mean training a large number of network combination. Different software environments were analyzed for network training. DIGITS from NVIDIA [4] was finally selected which is an excellent tool for network training and analysis. It is an open source tool and uses Caffe. It can create training validation and test sets as per the user need. It also has prebuilt sets for Imagenet, Cifar-10, MNIST and various others. It also has pre built popular CNN networks such as Lenet and VGG.

It also creates a model file called `.prototxt` and a `.caffemodel` file. These files can directly be ported to Xilinx Vivado tool for full precision CNN mapping which has taken abstraction to a very high level. Caffe model weights can also be modified to 16 bit precision file [30]. However, they cannot be modified to 1 or 2 bit for which a separate training scheme was used. In any case, for network study, this tool proved to be sufficient. The web based GUI environment is shown in figure 19.

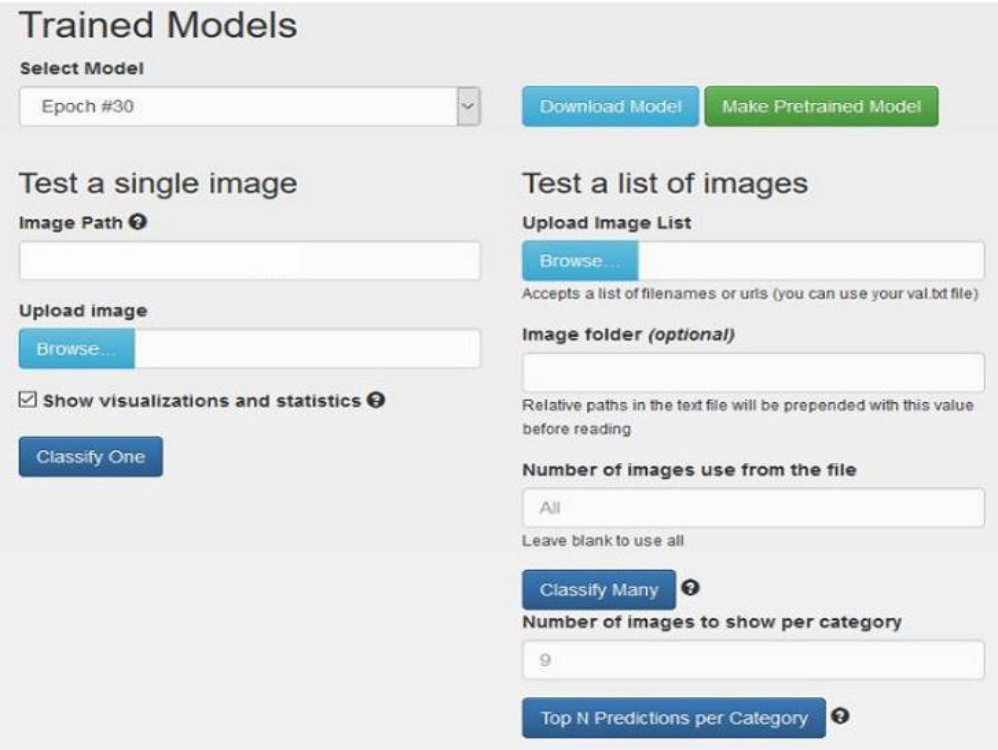


Figure 19: Image Showing Interface of Training Tool

4.2 Architectural Implications on CIFAR-10

As explained in the previous sections, CIFAR-10 is a mid sized data set with 10 classes of real life subjects like animals, vehicles etc. There are a number of convolutional neural networks proposed for classification of different types of data sets. The possible combinations of neural network architectures are infinite and hence present a huge design space challenge. Combined with the possible options for hardware implementation, this starts to become an even bigger problem. There have been efforts from both software and hardware end to understand the optimal solution to proposed design space problems. Roofline model is one such technique used by Zhang et el. [9] to understand memory vs compute management.

One popular FPGA implementation of CIFAR classification architecture was presented by [1] which presented a network called Binarized Neural Network. It was a combination of conv-conv-pool layers repeated three times followed by three FC layers. This is shown in table 3 adopted from the original work. *Input Fmaps* defines the number of input layers going in to each layer. For example, in Conv1, number of input layers is 3 since the it is colored image (r,g,b layers). *Output Fmaps* show the number of outputs from each layer. This also means the number of filters in each layer since number of outputs is equal to the number of filters. In case of Conv1, the architecture has 128 filters. *Output Dim* shows the size of the output. In case of Conv1, the output image is

32x32. For a fully connected layer (FC1-3) in table 3, *Input Fmaps* means the total number of neurons in the input layer of the FC component. Similarly *Output Fmaps* would mean number of neurons in output layer. *Output Dim* of 1 indicates it has one dimensional output of *Output Fmap* number of neurons.

Table 3: Network Architecture Proposed in [1]

Layer	Input Fmaps	Output Fmaps	Output Dim
Conv1	3	128	32
Conv2	128	128	32
Pool	128	128	16
Conv3	128	256	16
Conv4	256	256	16
Pool	256	256	8
Conv5	256	512	8
Conv6	512	512	8
Pool	512	512	4
FC1	8192	1024	1
FC2	1024	1024	1
FC3	1024	10	1

As evident from table 3, the combination of convolution, pooling and FC layers could be of any number. It is difficult to find a systematic approach to the problem of optimal design for dataset classification in literature. Hence, this work presents a brief study of convolutional neural network architecture and its implementation on classification performance.

4.3 Problem Statement

We formalize our problem statement in the form of following possible optimization points.

1) Impact of depth of network (number of convolutional layers)
2) Output feature map dimensions
3) Output feature map layers (number of kernels)
4) Filter Size
5) Impact of fully connected layers
6) Pooling layers
7) Other factors.

We shall now see how some of these factors impact classification of CIFAR-10 data set and explain the reason behind it.

4.3.1 Filter/Kernel Size

We have explained a few CNN architectures in the preceding sections. We see that different kernel sizes have been used by different architectures to get the maximum classification scores. The exact impact of varying the filter size on classification was hypothesized but not substantiated before. We explore the impact of different filter sizes on CIFAR-10 data set.

We take the following possible sizes of filter:

1) 3x3
2) 5x5
3) 7x7

We take the following approach to study impact of filter size on classification.

1) We take the number of outputs (filters) in the layer as 8
2) We fix number of input layers as 3
3) We fix number of convolutional layers as 1
4) We fix number of fully connected layers as 1.

It is also important to mention here that the filter size has a direct impact on the output map size. We pad the output maps with required number of zeros to make the size of output equal in each case. This is done to ensure equal size of the fully connected layer later on in the network.

The network architecture is shown in figure 20.

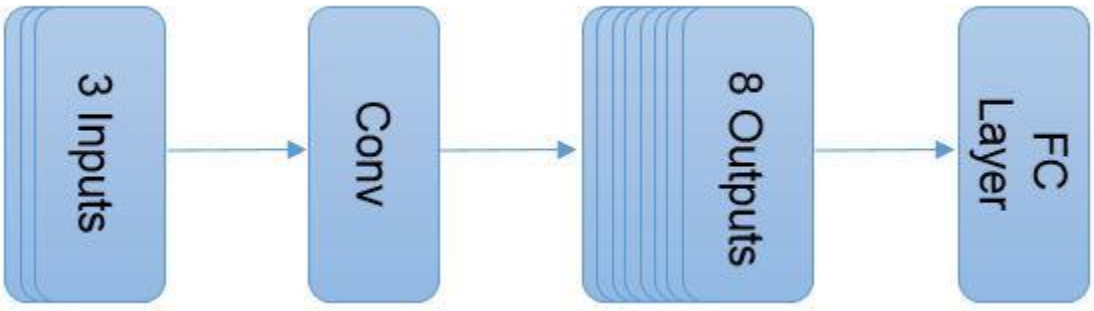


Figure 20: Architecture of Network Under Test

In Digits, we select CIFAR-10 dataset, input our architecture given in figure 20, set the number of epochs to 50 and run the model for 3x3, 5x5 and 7x7 filter size. Figure 21 shows training output and table 4 summarizes the data.

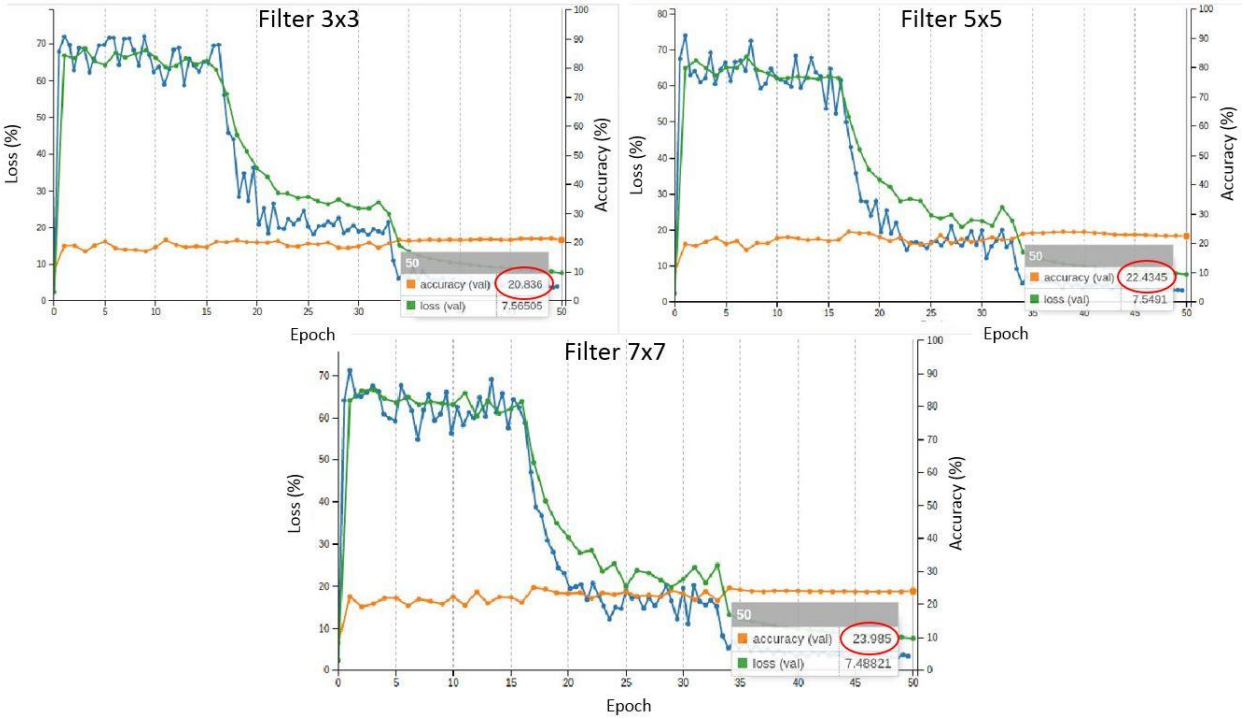


Figure 21: CNN Performance with various Filter/Kernel Size

In figure 21, x axis shows the number of epochs. Left y axis shows the loss value which is the sum of all errors in the training set. Right y axis shows the accuracy of the trained model. Accuracy shows the performance of the trained model. After 50 epochs, classification accuracy of 5x5 filter (22.43%) is greater than 3x3 filter (20.83%). Similarly, 7x7 filter performs better than 5x5 filter. We changed the number of filters to see impact of increasing filter size. We take a note here that increasing number of outputs will also be explained separately in the next section. Continuing our discussion on filter size, we see trend in classification accuracy with increasing filter size and number in figure 22. From figure 21 and 22, we see a rise in classification accuracy with increasing filter size regardless of the number of filters used.

We also observe a more substantial increase in accuracy for a filter size of 7x7. This would suggest a choice of a large filter to get better results for CIFAR-10. We hold on to this point until later sections where we will see the impact on hardware memory and finalize the architecture based on performance/memory tradeoff.

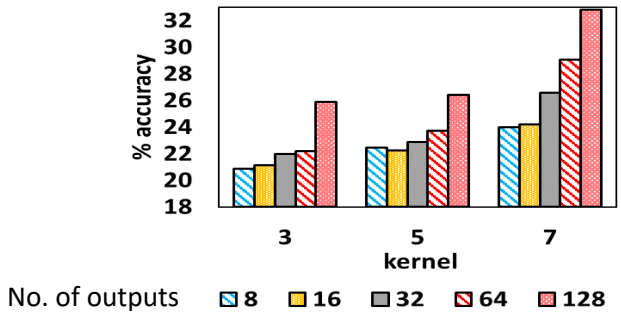


Figure 22: Comparison of Classification Accuracy with Filter Size and Number of Outputs

4.3.2 Output Maps

We change the number of output maps or the number of filters in the convolution layer to see the impact of filter numbers on classification. We tabulate the data in table 4.

Table 4: Network Performance in Relation with Network Configuration

conv layers	input maps	kernel	output maps	%accuracy
1	3	3x3	8	20.836
1	3	3x3	16	21.139
1	3	3x3	32	21.97
1	3	3x3	64	22.185
1	3	3x3	128	25.903
1	3	5x5	8	22.435
1	3	5x5	16	22.259
1	3	5x5	32	22.874
1	3	5x5	64	23.697
1	3	5x5	128	26.422
1	3	7x7	8	23.985
1	3	7x7	16	24.176
1	3	7x7	32	26.574
1	3	7x7	64	29.03
1	3	7x7	128	32.79

We see from the data in table 4 that there is an increasing trend with increasing number of filters. This is intuitive considering more data from filters should mean better results. Figure 23 helps us visually recognize this trend and summarizes both the impact of the filter size (marked in red, blue and green colors) as well as the impact of increasing filter numbers on the x axis.

We observe another pattern from figure 23. For filter sizes of 3x3 and 5x5, the performance curve is relatively flat up until the number of output maps cross 100. Even at that point, there is only an increase of around 3% in the performance of the classifier (table 4 for exact values). This is another

design consideration which needs to be taken into account while allocating hardware resources to the architecture since these are much sparser for a platform like FPGA as compared to a GPU.

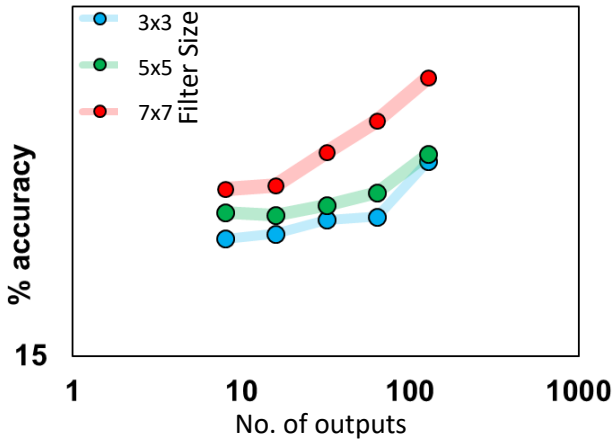


Figure 23: Increase in Classification Accuracy with Increasing Number of Output Maps

4.3.3 Fully Connected Layers

We studied the effects of increasing the number of fully connected layers in our network. For this study we again fix the number of convolution layers as well as the filter size, changing only the number of FC layers. In this, the first FC layer generates 1024 outputs. The second layer then generates 10 outputs for 10 classes. The filter size is fixed as 3x3 and the number of convolution layers are fixed as one.

We verify the results by changing the number of outputs from the convolution layers and see if the trends hold across varying number of input neurons of the fully connected layers (they depend on the output pixels of the convolution layer). The architecture format is shown in figure 24.

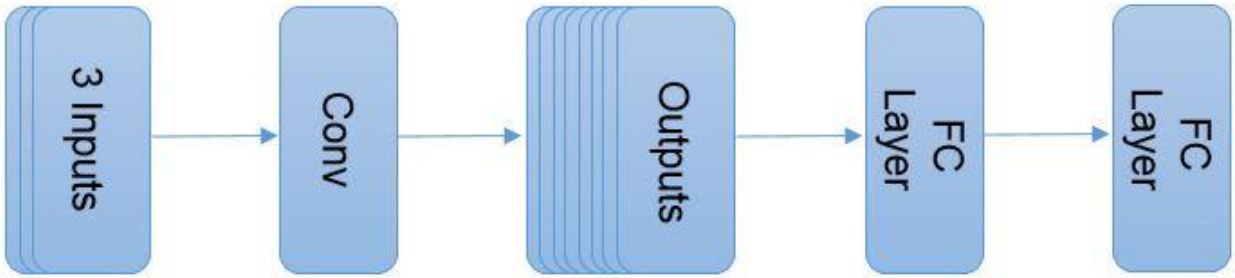
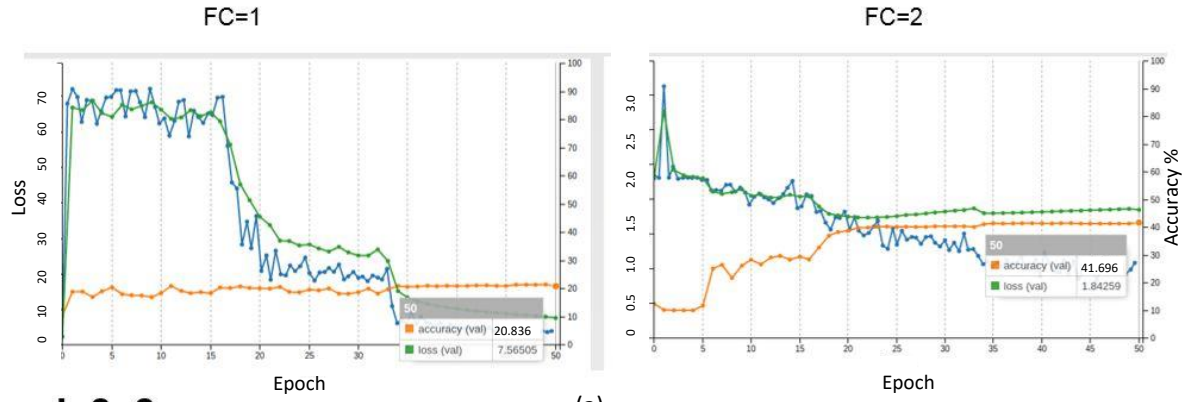


Figure 24: Architecture of Network Under Test with 2 FC layers

We run training on DIGITS again. The training results are shown in figure 25 and discussed below.

Kernel=3x3

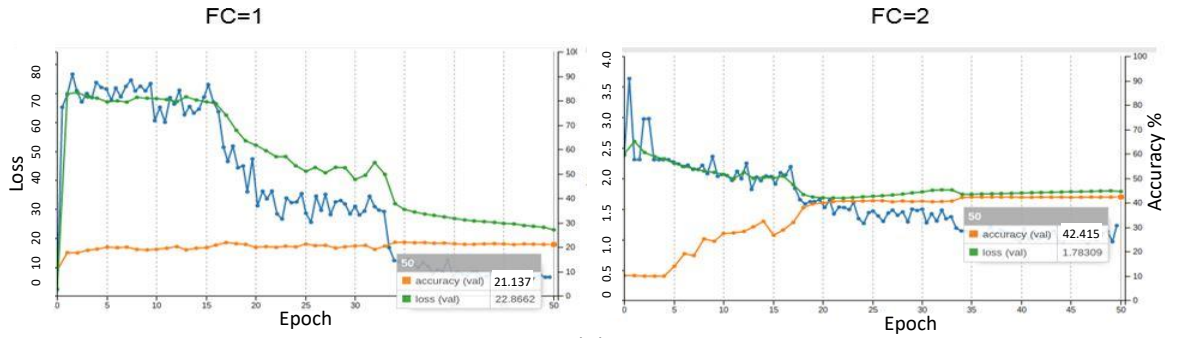
Outputs=8



(a)

Kernel=3x3

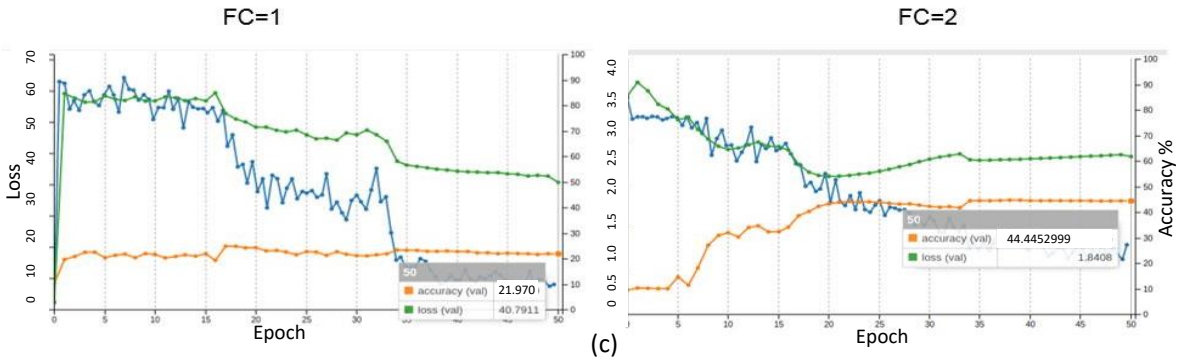
Outputs=16



(b)

Kernel=3x3

Outputs=32



(c)

Figure 25: Network Performance with Increasing Number of FC Layers for Various Filter Size

In graphs of figure 25, x axis shows the number of epochs. Left y axis shows the loss value which is the sum of all errors in the training set. Right y axis shows the accuracy of the trained model. Accuracy shows the performance of the trained model and we look at this in detail.

In figure 25 (a), filter size is fixed at 3x3 and number of outputs from convolution layer are 8. Left graph has a single FC layer while graph on right has two FC layers. We can see that the performance of model with 2 FC layers (41.69%) is better than model with single FC layer (20.83%).

In figure 25 (b), filter size is fixed at 3x3 and number of outputs from convolution layer are 16. Left graph has a single FC layer while graph on right has two FC layers. Again, performance of model with 2 FC layers (42.41%) is better than model with single FC layer (21.13%) for this case as well. We see a similar trend for figure 25 (c) as well which has 32 outputs from its convolution layer.

We see that performance of network with 2 FC layers is consistently better than model with a single FC layer. If we observe the three models with a single FC layer, we see that their performance increase is nominal with increasing number of outputs (20.83-21.97%). Similarly for models with 2 FC layers, increasing number of outputs has a nominal impact when compared with increasing number of FC layers. These findings are consistent with our results in the previous section regarding impact of increasing outputs on classification.

4.3.4 Convolution Layers

We move towards the impact of increasing convolution layers. Instead of simply increasing the number of convolution layers, which should intuitively improve performance, we wanted to study the impact of the convolution layers in a bit more detail.

For this test, we varied the number of filters in layer 1 from 8 to 32. Filter size was fixed at 3x3. We fix the number of fully connected layers to 1. Number of filters in conv layer 2 were changed from 16 to 64. We start off with a single pooling layer after conv layer 1. We visualize the architecture in figure 26 and tabulate results in table 5.

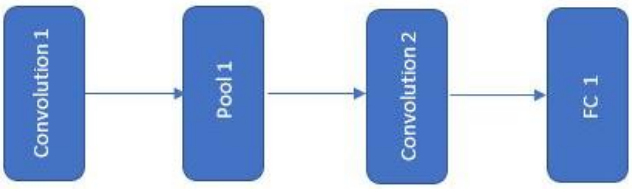


Figure 26: Architecture of Network Under Test with 2 Conv Layers

Table 5: Network Performance in Relation with Increasing Neurons in FC Input

layer 1 outputs	layer 2 outputs	Neurons in FC	%accuracy
8	16	4096	50.5
8	32	8192	34.27
8	64	16384	33.83
16	16	4096	50.86
16	32	8192	26.46
16	64	16384	31.96
32	16	4096	51.1
32	32	8192	28.18
32	64	16384	31.82

We fix the number of outputs from convolution layer 1 to 8. We vary the number of outputs from convolution layer 2 (16-64). We train the models and look at their performance accuracy. From first three rows of table 5, we see that accuracy decreases on increasing the number of outputs from conv layer 2.

Similarly for 16 outputs from conv layer 1, we see that on increasing the number of outputs from conv layer 2 (16-64), we get a lower performance. This trend is consistant for 32 outputs from conv layer 1 as well. We see in table 5 that even though increasing the convolution layer improves accuracy, increasing number of outputs from convolution layer 2 which feeds into the fully connected layer does not have a positive impact on the accuracy. One possible explanation for this could be data over fitting which results from too much data going into the single convolution layer

If we compare our results in table 5 with those in figure 25, we see that increasing the number of convolution layers has indeed had a positive impact on clasification accuracy. For model with 2 conv layers, worst case classification accuracy with 8 layer 1 outputs was 33.83%. for model with a single conv layer (fig 25), this accuracy was 20.83%. We observe this trend for 16 and 32 outputs from conv layer 1 and see that worst case for model with 2 conv layers is always better than model with a single conv layer. We visualize this in figure 27.

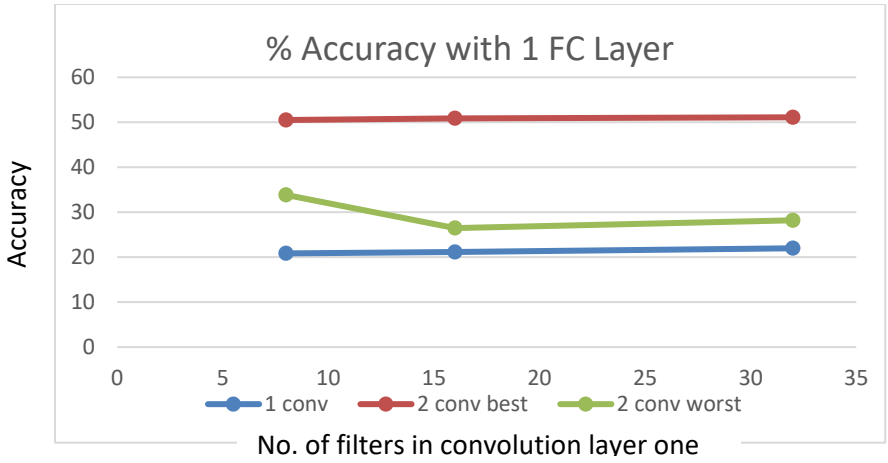


Figure 27: Comparison of Performance with Increasing Number of Convolution Layers

We visualize the impact of increasing number of outputs from conv layer 2 on the classification accuracy. In figure 28, x axis shows an increase in number of outputs from conv layer 2. Blue, red and green lines show conv layer 1 outputs of 8, 16 and 32 respectively.

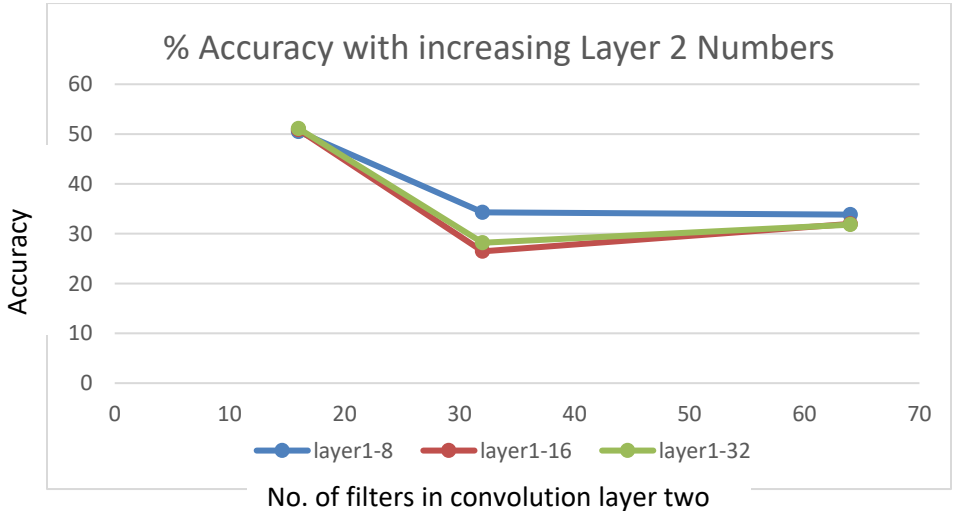


Figure 28: Performance Analysis with Increasing Outputs from Conv Layer 2

From figure 28, we see that for lower values of outputs from layer one and layer 2, we get the highest classification accuracy. Increasing layer 2 outputs has almost the same impact for all values of layer one outputs. We can see here that layer 2 is the dominant factor. This would again substantiate the claim that high number of layer 2 outputs which result in high number of neurons in the single fully connected layer has a negative impact on the classification.

In the above experiments, we had a single pooling layer. We now add another pooling layer after convolution layer 2 and see if reducing data going into the fully connected layer has any impact on the results. We can visualize the architecture from figure 29.

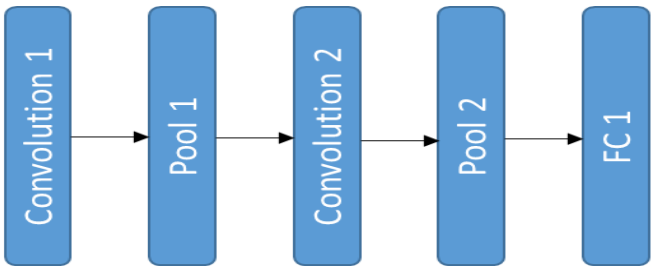


Figure 29: Architecture of Network Under Test with

We see the performance of the architecture in figure 30. We plot 3 curves for 8,16 and 32 outputs from convolution layer1. Y axis shows classification accuracy and we increase outputs from convolution layer2 along X axis. We get the best performance now at higher filter numbers in convolution layer 1 (16) and convolution layer 2 (32). Pooling layer after conv layer 2 reduces data going into the fully connected layer making performance better. Comparing curves in figure 30 with those in figure 28, we see that there is a performance improvement on adding a pooling layer. For Conv1 with 8 outputs, we have an improvement of over 10% for best case of both architectures. Similarly for Conv1 with 16 and 32 outputs we have an improvement of 12% and 1.5% respectively. On increasing the filter numbers in Conv2 beyond 32, we see a drop in classification accuracy which could again be explained by the large number of inputs going into the single fully connected layer. Over fitting of model due to large number of neurons at input of fully connected layer could explain why we get performance degradation on having too many filters or absense of pooling layer. Over fitting is usually taken care of at training level by using a *drop out function.* What this function does is that it ignores or considers each neuron of the fully connected layer by a probability *P* at both the forward pass and the backward pass. During test phase, we reduce the neurons by a factor *p* to account for the missing activations during training. Since we intend to implement the architecture on a fixed hardware, we had not incorporated the drop out part. Hence we could explain the impact on performance to too much data going in to the network which is not being dropped.

We run one more test to observe number of neurons in input of FC layer and its impact on classification accuracy. We vary input neurons in the fully connected layer by using pooling layer after conv 2 layer. We plot three graphs for 8,16 and 32 outputs from convolution layer 1 shown in figure 31. Y axis in all three graphs show the classification accuracy and the number of outputs from convolution layer 2 are increasing along the x axis.

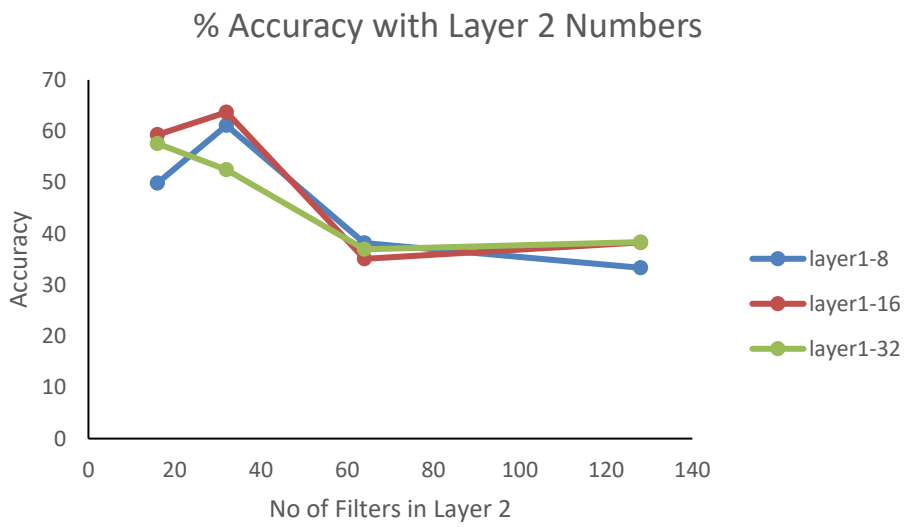


Figure 30: Performance Analysis with Increasing no. of Filters in Layer 2 and Layer 1

We take a closer look at figure 31(a). Number of outputs from convolution layer1 are 8 and outputs from convolution layer 2 are varied from 16 to 64. With 16 outputs from Conv2, we get 4096 and 1024 pixels from output of model without and with pooling respectively. Observing the plots (yellow line for pooling and grey for model without pooling), we get similar classification performance at this point (~45%). On further increasing the number of outputs to 32, we get 2048 pixels for pooling and 8192 for model without pooling. At this point, we observe that the model with pooling outperforms model without pooling (~22% difference in performance). Moving further in the graph, we increase the number of outputs from Conv2 to 64. We get 4096 neurons for pooling and 16384 neurons without pooling. We again observe that the yellow curve is higher than the grey curve by a small margin. However, we can see that the yellow curve (pooling layer model) has taken a dip while the grey curve remains constant.

Taking our experiment further, we change the number of outputs from convolution layer 1 to 16 and 32 in figure 31 (b) and figure 31 (c) respectively. We observe curves for models with and without pooling with increasing number of outputs and neurons. We see a very similar trend in both these graphs as well. Pooling layer has higher curves for all these cases. We also observe the reduction in performance post 2048 neurons.

From our discussions regarding results obtained in figure 30 and figure 31, we have observed performance increase due to reduction in input neurons in the fully connected layer by using pooling layers in the architecture. Hence, we should incorporate pooling layers to compensate for our lack of drop out in training and avoid over fitting in the model.

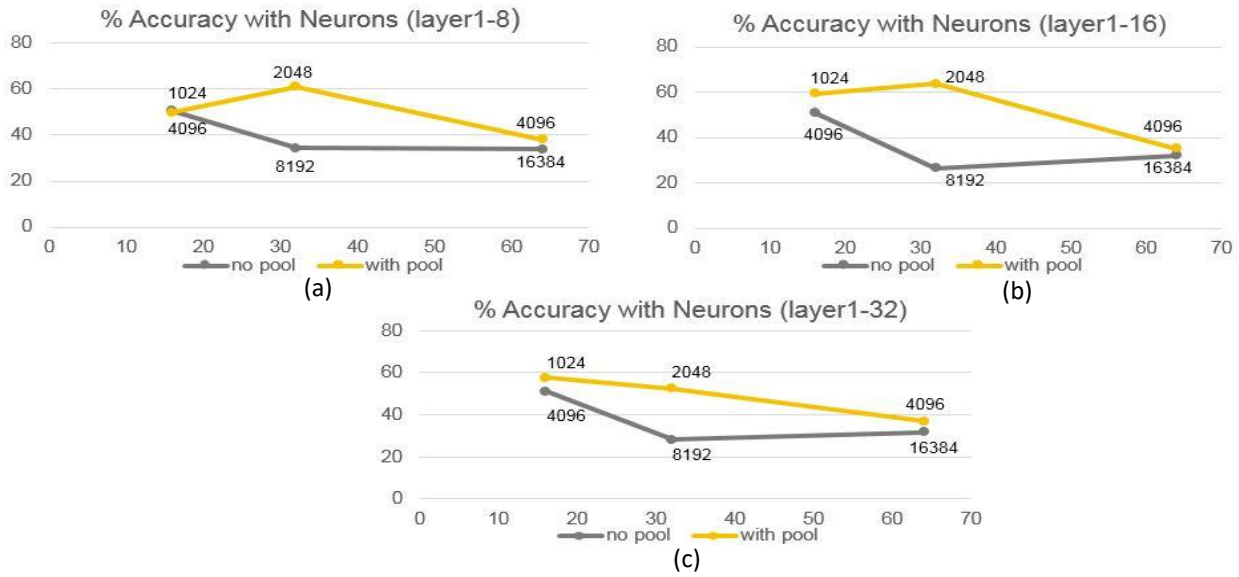


Figure 31: Performance impact of increasing data generated by convolution layers

4.3.5 Concluding Remarks

We form a general idea of the nature of the CNN architecture for CIFAR-10 classification. We conclude that any architecture designed should be deep with multiple convolution layers for better results. The depth of the network is another design decision based on the hardware resources available. We can also state that 2 fully connected layers outperform a single layer. Another positive outcome of having more layers is that we can gradually reduce the number of neurons between layers instead of directly converting to 10 neurons at output which can cause significant performance issue as seen in the previous section on neurons and its impact on classification.

We can also state that strategic placement of pooling layer periodically in the network help to reduce redundant data and over fitting and hence we will be doing that. It also explains the presence of pooling layers in so many high performing architectures which have recently come up by various research groups.

We have seen that larger filter size has better classification. However, this improvement is not substantial considering the cost (147 weights for 7x7x3 filter compared to 27 weights for 3x3x3 filter). Hence we are inclined to use a smaller filter size in favor of memory conservation.

4.4 Binary weighted CNN

In the previous section, we discussed some of the design parameters required for an efficient convolutional neural network design and how each of these impact the performance of the network. We got a good sense of which parameters were important and are now in a more informed position to make tradeoffs between resource allocation and performance for each layer.

We make the following decisions for our deep network for CIFAR-10 classification.

4.4.1 Convolution Layers

We want to make our network deep. As shown in the previous section, increase in convolution layers has a marked impact on the performance of a network. We also do a quick memory consumption analysis. For each convolution layer with 50 inputs and 50 outputs, we have 50 filters of size *'a'* x *'a'* x 50 where *a* is length of one dimension of filter. For instance if we select filter size as 3x3, we get memory consumption for one convolution layer in table 6.

Table 6: Resource Requirement of 3x3 Binary Filters

Filters	**Dimensions**	**Depth**	**Weights**	**Memory (1–bit)**
1	3x3	50	3x3x50	450 bits
50	3x3	50	3x3x50x50	22.5Kb
128	3x3	128	3x3x128x128	147.5Kb
256	3x3	256	3x3x256x256	590Kb

We had already shown in previous sections that most of the memory elements are concentrated in the fully connected layers and not in the convolution layers. Looking at table 6, we observe that memory consumption in convolution layers is limited to a few hundred Kb even for very large number of filters. Looking at resource requirements of convolution layers, we decide to limit the number of convolution layers to 6 at this point. This is a judgement based on the hardware resources available for implementation. This would be further clarified in subsequent sections.

4.4.2 Convolution Layer Outputs

We look at table 6 again. We see an increase of our 300% in memory consumption when number of outputs are increased from 128 to 256. We also get a similar ratio of increase in number of computations as well. Hence, we limit our outputs to 128. This was done since increasing number of outputs per layer did not show a marked increase in performance. Again this was a design decision based on analysing the tradeoff between resource utilization and increase in classification performance. We decide to have 6 convolution layers with the following outputs.

1) Conv Layer 1: 64 outputs
2) Conv Layer 2: 64 outputs
3) Conv Layer 3: 128 outputs
4) Conv Layer 4: 128 outputs
5) Conv Layer 5: 128 outputs
6) Conv Layer 6: 128 outputs

4.4.3 Filter Size

We had shown that increasing the filter size had an increase on the classification performance of the network. Let us look at the impact of increasing filter size on memory consumption as well as the number of operations in the convolution layer in table 7.

Table 7: Comparison of Resource Requirement in with Filter Size

Filter	Dimensions	Memory	Operations
1	3x3	9 bits	9 XNOR/count
2	5x5	25 bits	25 XNOR/count
3	7x7	49 bits	49 XNOR/count

There is more than 500% increase in going from a 3x3 filter to a 7x7 filter. We had seen that the increase in classification accuracy by increasing the filter size is dwarfed by the increase in performance by adding more layers. For example, with a filter size of 3x3, we get a classification performance of 20.8%. On increasing the number of FC layers, we get an increase of over 20% while increasing convolution layers to 2 given an improvement of around 30% as can be seen from figure 25 and figure 27. Compare this to increasing the filter size to 7x7 which gives a performance improvement of around 3% as shown in figure 21. Since filter size greatly adds to the number of

operations in the convolution layers, we would like to limit it to a smaller size and compensate it through greater number of convolution layers. Hence we make another design decision and pick the filter size as 3x3 in our design.

4.4.4 Pooling Layers

We had seen the negative impact of having too many input neurons at the fully connected layers. Hence we intend to limit the number of neurons going into the first layer. A good way of doing this without losing out on the effectiveness of the convolution layers is to use the pooling layers. Hence we add pooling layers in our design. We saw a degradation in performance on increasing neurons more than 2048 in the previous section and want to limit to 2048 the input neurons in the FC layer based on the experience from the previous training exercise. For us to achieve this number with 128 outputs from the last convolution layer, we calculate the required number of pooling layers. For 128 outputs coming out of the last convolution layer and going into the fully connected layer, we need to have *'2048/128'* outputs in each output. This number comes out to be 16. Hence the outputs from last conv layer need to be size 4x4x128 which generate 2048 neurons on the FC layer input. This has also been summarized in table 8.

Table 8: Relation of Input Neurons with Filter Dimensions

Input image size	Neurons in FC1	Output Maps in C6	Output Map Pixels	Output Map Size
32x32	2048	128	2048/128=16	4x4

Since we are considering CIFAR-10 data set which has images of size 32x32, we need to reduce the size to 4x4 before going into the FC portion of the network.With a pooling layer of size 2x2 and step size 2, we reduce the the size of image by half after each pooling layer. Hence we have a logarithmic relationship here. To reduce the input from 32 to 4 we have a reduction by a factor of 8. Hence we need $log_2(8) = 3$ pooling layers in our design. We decide to place these layers after every 2 convolution layers. This is because we want to extract the maximum information from a given map size before dropping pixels and discarding information.

4.4.5 Fully Connected Layers

So far in the design, we have managed to save resources by using filters of smaller size in the convolution layers. Using regular pooling operation in the design also reduces memory consumption in each subsequent layer. We use these savings in memory to add more fully

connected layers in the architecture. Figure 25 showed a drastic increase in classification performance on adding fully connected layers to the network architecture. Also since they do not have a lot of overhead on the number of operations as shown in figure 14, while mostly impacting the memory consumption, we add 3 fully connected layers in the design. We look at the memory consumption in these layers in table 9.

Table 9: Fully Connected Layer Dimensions

Layer	Inputs (a)	Outputs (b)	Weights (axb)
FC1	2048	1024	2097 Kb
FC2	1024	1024	1048 Kb
FC3	1024	10	10 Kb

4.4.6 Final Configuration

We have the following final configuration of our network.

1) Conv-1
2) Conv-2
3) Pool-1
4) Conv-3
5) Conv-4
6) Pool-2
7) Conv-5
8) Conv-6
9) Pool-3
10) FC-1
11) FC-2
12) FC-3
13) Softmax

Final configuration has been illustrated in figure 32. We have input of size 32x32x3. Conv1 has 64 filters and produces 64 outputs of size 32x32x64. This goes into Conv2 which also has 64 filters and produces outputs of size 32x32x64. Pool1 reduces the outputs to 16x16x64. Conv3 and Conv4 each have 128 filters. Output from Conv4 has dimensions 16x16x128. Pool2 again reduces the size to 8x8x128. Conv5 and Conv6 also have 128 filters each and produce output of 8x8x128. Pool3 reduces size to 4x4x128. FC layer1 has 2048 neurons at input and produces

1024 outputs. FC layer 2 has 1024 inputs and 1024 outputs. Final FC layer takes 1024 inputs and gives 10 outputs for one class each. Probabilities for each class are then calculated in Softmax.

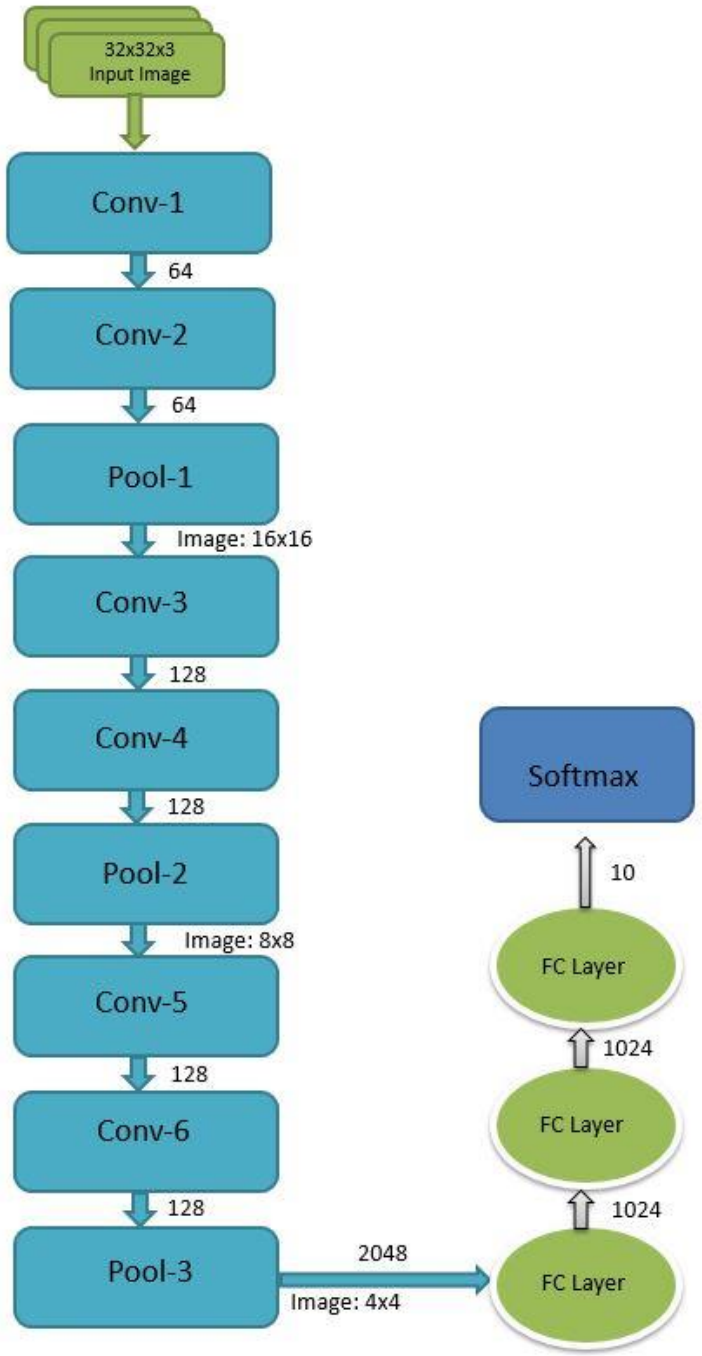


Figure 32: Final Network Architecture to be implemented.

4.5 Binary CNN Training

While training the full precision network, we used DIGITS from NVIDIA. It is an excellent tool which provided us with easy network configurations, training set preparation, real time performance monitoring and even a configuration file to be imported on to the FPGA. However, it does not provide us with any method to configure bit width in the system, neither are the training weights reduce-able to a single bit width without significant loss in precision.

One of the most widely used algorithms used for single bit weight training has been proposed by Courbariaux et al. [5, 6]. The algorithm restricts the training weights to either -1 or 1. These can be reformatted to represent +1 by 1 and -1 by a zero.

The algorithm is available online and is available for public use [6] and we shall briefly explain it.

4.6 Binary Training Algorithm

We look at the binarization function as detailed in [6] to get the binary valued weights.

4.6.1 Deterministic vs Stochastic Binarization

Traditional backpropagation algorithm gives real valued weights for the trained network. These need to be converted during training phase to binary values. We can do this in two different ways. The first is to have a deterministic function given by equation:

$$x^b = Sign(x) = \begin{cases} +1 \; if \; x \geq 0, \\ -1 \; otherwise \end{cases}$$

where x^b is a binary valued variable and x has a real value. This function looks at the sign of the input and makes it either +1 or -1 based on the sign. The second way to binarize a value is stochastic which is given by equation:

$$x^b = \begin{cases} +1 \; with \; probability \; p = \; \sigma(x), \\ -1 \; with \; probability \; 1 - p \end{cases}$$

Where σ represents a hard sigmoid function which is given as:

$$\sigma(x) = \max(0, \min(1, \frac{x+1}{2}))$$

Stochastic algorithm requires generation of random values which is harder to achieve and hence the algorithm given in [6] use a deterministic method.

4.6.2 Gradient Computation

During the training process, real valued weight updates are stored and kept track of before applying the deterministic binarization function. During forward pass of training, error value is calculated for each training input and is accumulated. This is the gradient which is used to update the weight values in the backward pass. These error values are not binarized but are stored as real valued number with sufficient precision. Once the weight has been updated using the gradient, the weight values are binarized using the deterministic binarization function.

4.6.3 Algorithm Forward Pass

The algorithm uses mini batch of training data. A mini batch is the set of training data fed to the network after which the weight update is applied. It goes through the following steps:

1) Initialize network weights W_k where k ranges from 0 to number of weights.
2) Convert weights to binary $W_k^b \leftarrow Binarize\ (W_k)$
3) For each input , accumulate updates by multiplying a_k^bwhich is the binarized difference of training data input and output. Update is calculated as $\delta_{k+1} \leftarrow a_k^b W_k^b$
4) Apply batch normalization to real valued update.

4.6.4 Algorithm Backward Pass

Once weight updates are calculated in the forward pass, update is applied during the backward phase of the algorithm. It performs the following steps:

1) Calculate update to apply $update = \delta_k\, W_k^b$
2) Apply update to network weight $W_{k+1} \leftarrow update\ x\ W_k^b$
3) Apply batch normalization
4) Binarize weight $W_{k+1}^b \leftarrow W_k$

4.6.5 Results

In [6], the algorithm was used to train binarized neural network for CIFAR-10 dataset. The network achieved an error rate of 11.4%.

4.7 Binary CNN Training Setup

The publically available library was downloaded and set up on the GPU work station. Anaconda environment was setup with the downloaded library to run the training algorithm. The library uses older versions of python libraries and needs the appropriate versions of the libraries to work.

Use of Anaconda makes the management of libraries easy to handle. The required list of python libraries needed to run the algorithm are shown in table 10.

Table 10: Python libraries required to train binarized convolutional neural network

Library	Version
Lasagne	0.2.dev1
Libgpuarray	0.7.5
Mako	1.0.7
Mkl	2018.0.0
Nose	1.3.7
Numpy	1.13.1
pygpu	0.7.3
Pylearn2	0.1.dev0
Theano	0.8.0
Six	1.10.0

With the required envirnoment set up, network shown in figure 32 was implemented in Theano framework. The network was trained for 400 epochs on the GPU which took around 2 days of training time. The results are given in table 11.

Table 11: Classification error rates for trained network

Error Type	Error%
Validation Error Rate	14.66%
Test Error Rate	14.9%

Validation error is the error calculated from validation set which is a subset of the entire CIFAR-10 dataset. Test error is the primary measure of performance and is calculated from test set which is also a subset of the entire data set. 10000 images were used for training and test subsets each.

The classification accuracy achieved for this network is around 3% less than the best performance achieved in the original work [6].

CHAPTER 5

IMPLEMENTATION HARDWARE AND SOFTWARE OVERVIEW

5.1 Introduction

We will now begin to discuss the hardware platform for the implementation of the binarized convolutional neural network. We will look at the current trends in FPGA capabilities. We will also address FPGAs being used by other researchers in similar domain. We will discuss the viability of FPGAs for the implementation of reduced precision networks and its advantage over using a GPU. We will also briefly discuss the Zynq SOC architecture [31] which is used in this work for the implementation of the network.

5.2 FPGA Capabilities

FPGAs have recently seen a remarkable increase in on chip resources and processing capabilities. This has also driven renewed interest in implementing highly parallel architectures on FPGAs and utilize their inherent parallel processing power and reconfigurability. This had limited scope before due to limited memory resource available.

Table 12 summarizes the performance of some of the latest FPGA devices.

Table 12: Resource specifications of latest FPGAs.

FPGA Device	CLBs	DSPs	Total Memory
Arria10	1150k	3046	53 Mb
Virtex 7	693k	3600	53 Mb
Stratix V	622k	256	50 Mb
Stratix V2	695k	3,926	50 Mb

We can see from table 12 the substantial on chip memory available on the FPGA fabric. This is usually in addition to the off-chip memory which can range to several Gigabytes connected to these chips. On chip memory is however a more critical factor since it is less affected by other factors such as memory bandwidth available on the bus which can potentially nullify the gains of parallel computing through FPGA.

5.3 FPGA vs GPU

GPUs have a very high processing power, as high as 11 TFLOPs/sec (floating point operations per second) [32], making them highly suitable for networks such as CNN. However, their architecture is highly optimized for floating point arithmetic which they can perform very efficiently. When we speak of low precision networks, GPUs have no way of dealing with them other than to treat them as a full precision number. Recently, a tool called Ristretto [26] has been introduced which reduces data width to 16 bit from 32 bits for networks trained on Caffe using GPUs and hence have some potential do utilize low data width. This however, still pales in comparison to a flexible architecture like an FPGA. FPGAs are highly customizable. They can be configured for any user defined data width ranging from a low end of 1 bit to the high end of up to 1024 bits in some of the larger boards.

Table 13 shows some of the implementations of CNN which have made use of the FPGA customizability and their performance. The table shows the CNN architecture modeled in each of the research works. It also states the dataset used for performance analysis. Implementation platform and maximum performance (GOPS) achieved by each architecture is also given along with operating frequency, power consumption and data width used. From table 13, we see network data width ranging from 32 bits to 2 bits. 2 bit network [33] have shown promising results on even very large datasets such as ImageNet.

Table 13: FPGA Implementations of CNN

Paper	Architecture	Dataset	FPGA	GOPS	Frequency (MHz)	Power (W)	Precision
[34]	VGG16	ImageNet	Arria10	645.25	150	N/A	8/16 bit
[9]	Alexnet	ImageNet	Virtex 7	61.62	100	N/A	32 bit
[24]	VGG 16	ImageNet	Zynq	187.8	150	9.63	16 bit
[35]	Alexnet and Lenet	ImageNet	Virtex 7	222.1	100	24.8	8 bit
[36]	VGG16	ImageNet	VC707 & ZC706	316 On Zynq	170		16 bit

				1250 on V7			
[1]	Alexnet	ImageNet	VC709	565.94	156	30.2	16 bit
[33]	DoReforNet	ImageNet	Zynq	410.22	200	2.26	2 bit
[37]	Alexnet& NIN	ImageNet	Stratix V	114.5 Alexnet 117.4 NIN	100		8/16 bit
[35]	Alexnet	MNIST CIFAR-10	Stratix v	5905.40 MNIST 9396.41 CIFAR10	150	26.2	8 bit

Another major advantage of using FPGAs in place of GPUs is the power consumption. When we speak of embedded devices, we quickly realize the non-viability of GPUs due to their high-power consumption which is orders of magnitude more than even the biggest FPGA boards.

Hence, for these reasons, FPGAs present an ideal platform for implementing these algorithms.

For all the above stated reasons, we selected ZCU-102 board [31] from Xilinx which incorporates Zynq architecture [7].

5.4 Zynq-7000 Architecture

ZCU-102 development board from Xilinx incorporates Zynq-7000 system on chip (SOC) architecture. Zynq SOC has an on-board ARM cortex A-9 processor connected with the FPGA programmable logic via AXI bus interface [7]. The on-board ARM core can reach clock speeds of up to 1 GHz. It has over 3 Gigabits of off chip DDR memory which can be accessed by the programmable logic through one of five high performance AXI ports on the ARM core. The board has over 32Mbits of on chip BRAM which can be used for high speed memory access without going off chip for DDR memory access. The board has over 275k LUTs for combinational logic implementation. It has more than 500k flip flops and over 2500 DSP slices for floating point arithmetic implementation.

Figure 33 shows schematic of ZCU-102 board incorporating ARM coprocessor adopted from original datasheet from Xilinx [7].

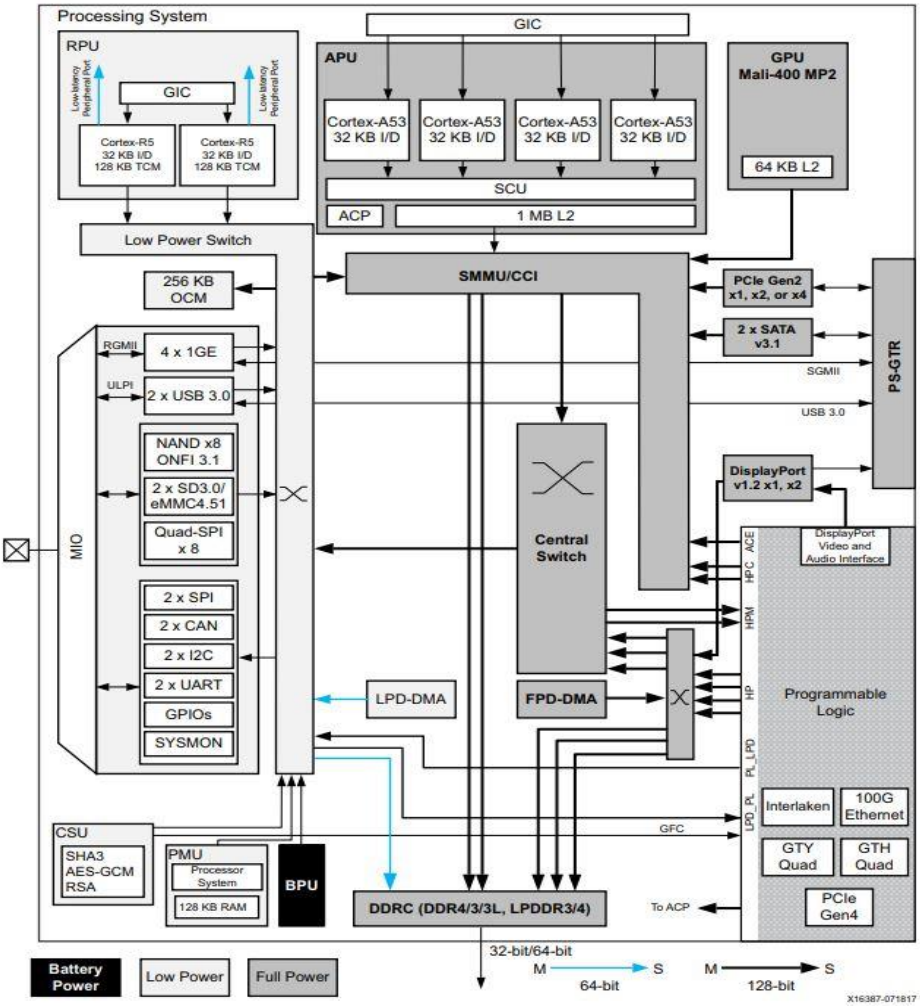


Figure 33: ZYNQ SOC Architecture Schematic [7].

The board incorporates a 64-bit quad core ARM Cortex A-53 [38] and a ZU-9EG programmable logic from Xilinx. The board supports several high speed interface ports such as USB 3.0, DisplayPort 1.2a as well as PCI ports which can be used to interface the board with an external display or computing platform as well. Figure 33 also shows Direct Memory Access (DMA) ports which are used to transfer data to the programmable logic. These are controlled through the ARM based processing system. It also has an on board graphical processing unit which can be used for compute intensive operations by the ARM core. It supports several different types of data transfer modes between the programmable logic and the processing system based on the user need. The board also supports several other peripheral interfaces such as timers to measure performance,

multiple clock support for different PS and PL operating frequencies, on board switches and buttons for external data input as well as several LEDs for status indication.

5.5 Software Overview

FPGAs like ASICs have traditionally been designed using Verilog or VHDL in one form or another. Verilog is a relatively low-level language and requires high level of expertise not just in programming concepts but also hardware paradigms.

While hardware definition languages such as those mentioned above provide great control to the designers at the gate level making very compact designs possible, it also introduces great challenges from troubleshooting and complexity point of view. With increased size and complexity of the design, troubleshooting becomes more and more complicated. Also, very large designs require a lot of skill and expertise in terms of area, timing and design analysis. Hence, this design and programming challenge has been a big hindrance when it comes to widespread adoption of even a flexible architecture such as an FPGA by programmers especially from a software background.

Xilinx has recently introduced a high level synthesis tool called Vivado HLS which offers great benefits in terms of project time and complexity [39]. Vivado HLS enables the designer to code in high level languages such as C, C++, OpenCL or SystemC. Not only does this drastically reduce required design time but also makes the design more manageable by utilizing high level data structures.

The tool generates a synthesizable Verilog code from the high level language code which can be ported to the Zynq Board. It also provides resource and timing analysis of the design which is very useful for early diagnosis of the design and can be used to make design improvement if necessary. However, there are still certain limitations in the system since the underlying processor is still a hardware platform and needs to be kept in mind while designing on the HLS tool. We discuss some of the capabilities and limitations of the tool which have to be considered while designing our CNN architecture.

5.6 VIVADO Components

The Vivado package from Xilinx is composed of the High Level Synthesis tool, Vivado design integrator, IP configuration tool and a software development kit. These tools work in tandem to complete all stages of the design flow.

We used C++ to code in the HLS tool. The process starts also includes writing a test bench in C++ which interacts with the HLS code for sending inputs and outputs and is used for testing and validation of the design. This is a very powerful feature which enables the following features:

1) Test of Software/algorithm

2) Synthesis Check

3) Software/Hardware co-simulation

Co-simulation feature in the HLS tool is used to compare the results of the software code and the high level code for synthesis and check the validity of the design. The tool runs the software design and stores outputs against given inputs. It then runs the high level code for hardware and similarly generates the outputs. It automatically compares the two outputs and gives a pass/fail result for the design. Hence in this way, we move systematically from checking algorithm towards checking the hardware design all packaged into a single tool and done using a high level language.

After satisfactory performance of the design is achieved, it is packaged in the form of an IP by the HLS tool. This IP is then used by the VIVADO block design tool to connect peripheral components to the main IP. In our design, we need the ARM core working with the FPGA core for data transfer as well as the implementation of Softmax. This core called the Zynq core can conveniently be added using the graphical user interface in the block design. We can also utilize programmable logic based MicroBlaze core which comes with the package. Similarly, instead of making a single large design, we can divide the CNN design into a number of smaller IP blocks and connect them together in the block design.

The graphical user interface in the block design even lets us connect memory such as block ram and DDR and make them shareable between different IP blocks. This will be very useful for our design which consists of convolution layers connected in series with each other, each sending the output as an input to the next layer.

To control the overall design, we will make use of the System on Chip (SOC) architecture of the Zynq board. The software development tool of the tool comes into play here. Vivado tool not only generates bit stream for the PL but also exports design to the software environment with necessary handles to send receive data and control the programmable logic.

We can make the ARM core as our main controller monitoring states of each of the IP block and hence the processes going on in the CNN network. It can be used to send data from one layer to the next as soon as it is done working on the inputs and hence maximize throughput of our design.

This can be done using the SDK tool which contains the required header files and libraries as part of the package.

We now discuss a few coding guidelines and compiler strategies which we will be using in designing our IP blocks for the CNN architecture

5.7 Coding Guidelines

We extract some important coding guidelines and compiler directives which will be useful for us for efficient implementation of our algorithm. These are all available in the extensive user guide provided by Xilinx [8].

These guidelines are also important because we are working in a domain which is in between software and hardware. It is easy to get carried away by the flexibility of a software environment without considering the underlying hardware that will be running that program. We will discuss the capabilities of the HLS tool and some important limitation which must not be violated during the coding process.

5.7.1 HLS Stream

HLS streaming library can be used in C++ coding in Vivado HLS for data transfer to and from the programmable logic. They can be used within a function or at the function definition to define method of input and output. Streams are translated on the RTL logic as FIFO blocks for data synchronization and transfer and can optionally be done using handshake signals based on user need.

Streaming is done using AXI protocol. It has the following modes:

1) AXI4 Burst Mode
2) AXI4 Lite Mode
3) AXI4 Stream Mode

Burst and lite modes are used to transfer data blocks. Stream is used for continuous transfer of contiguous blocks of memory. This is interfaced with the high performance port of the Zynq processor which can then be used to transfer data from DDR to programmable logic.

It can be used in the following way:

1) Include the hls_stream.h header file in the design
2) Declare the hls::stream interface.
3) Use HLS pragma to declare input or output as hls stream (pragmas are discussed below)

HLS streams can be transferred as 32 bit wide or more data packets. This can be even used to send smaller data width through efficient packing techniques of data into 32 bit wide registers and then unpacking on PL.

5.7.2 Pragma

Pragmas are a very powerful feature of the Vivado design suite and can be used in a number of applications. They are used in a wide ranging way from declaring port types to pipelining the design. We mention some important pragmas below:

5.7.3 Pragma Interface

Pragma interface is used to declare a port as a certain type of interface. This is necessary to inform the synthesizer as to what type of port to create on the RTL along with associated logic for the input and output ports.

We get the following syntax from Xilinx online help [40].

#pragma HLS interface <mode> port=<name> bundle=<string> \

register register_mode=<mode> depth=<int> offset=<string> \

clock=<string> name=<string> \

num_read_outstanding=<int> num_write_outstanding=<int> \

max_read_burst_length=<int> max_write_burst_length=<int>

The streaming mode we mentioned before is the “mode” section defined above. The mode can also be used to declare a “bram’ instead of the streaming interface if we intend to use the on chip memory. This is important for our design since we intend to use on chip memory for all operations. Important pragma interfaces are mentioned below:

1) Ap_none
2) Ap_fifo
3) Ap_memory
4) Ap_bram
5) Ap_axis
6) Ap_axilite

Details of all the ports and their usage can be found at [40]. Ap_none pragma implements a simple data port without any protocol.

Fifo pragma implements an active low fifo with empty and full ports. It is a standard fifo design with input and output ports. However, bidirectional read and write are not supported.

Memory pragma generates a memory block (RAM). It generates a RAM port on the block design which can be used to connect with external ram.

Bram pragma implements a similar memory port on the block design. This can be connected to a BRAM in the block design.

Axi stream and light interfaces are used for data streaming from off chip memory to the programmable logic.

5.7.4 Pragma Inline

This pragma works in the same was as it works in standard c++. It eliminates a function call to the called function and instead incorporates it into the calling function. This removes function call overhead. Similarly, the compiler can be explicitly instructed not to inline a function even if it seems to optimize the code by the compiler using the Inline Off directive. The syntax of using the pragma is given below:

#pragma HLS inline <region | recursive | off>

5.7.5 Pragma Pipeline

This pragma pipeline loop instructions and enables concurrent execution of statement reducing overall latency of the loop. The user can even define the initiation interval (II) between the pipelined instructions. There could be cases where the compiler fails to pipeline design due to issues such as dependencies or hardware resource constraints. In this scenario, the compiler takes the following actions:

1) Gives warning

2) Implements pipelined design with minimum possible initiation interval

The syntax used is given below:

#pragma HLS pipeline II=<int>

5.7.6 Pragma Unroll

Loop unroll pragma creates independent operations from a loop. At RTL level, it creates multiple copies of the same instructions to increase throughput of the design. By default design, the compiler generates rolled loops in RTL and the synthesizer generates RTL logic for loop control with number of iterations. In case of loop unrolling, independent RTL blocks are generated to execute each iteration. It also allows designer to specify the unrolling factor. For instance, if there are 10 loops, we can unroll the design by a factor of 2 to execute 5 iterations each. The syntax of using the pragma is given as:

#pragma HLS unroll factor=<N>

5.7.7 Dynamic Programming

Dynamic programming is one of the most powerful features of C++. It lets program allocate run time memory on heap and deallocate at the end to maximize resource utilization. However, since we are dealing with a hardware platform, this is strictly not allowed. Dynamic programming would effectively mean changing hardware fabric at run time. Hence this is not allowed.

5.7.8 Loop Iterations

Loop iterations in any given loop must be known at compile and synthesize time by the compiler. The program cannot change the number of iterations the loop will make at run time. This is because the compiler optimized and allocates resources along with loop control logic at compile time.

5.7.9 Object Oriented Programming

Vivado allows a wide range of OOP concepts to be utilized during programming. These include functions, classes, structs and even pointers. Care however must be taken as the compiler translates these blocks on to hardware logic by various means. For instance a lot of these constructs will be inlined in the function. Hence over usage of data structures on hardware can often lead to more problems than solutions.

CHAPTER 6
CNN HARDWARE DESIGN

6.1 Problem Formulation

In this work, we focus on the implementation of binarized convolutional neural network building blocks on Zynq Architecture. It is important to note here that the focus is not just to fit a deep learning algorithm on chip which can already be done more conveniently using vendor provided libraries. These libraries provide full precision implementation so far and we will be focusing on reduced precision binary net and its required building blocks.

We achieve the following goals:

1) Finalize design requirements for a working binarized convolutional network model
2) Model CNN building block algorithms
3) Define HLS design flow
4) Design building blocks
5) Optimize design for speed and power consumption

We also note here that the goal is not merely to design a working model but also understand the optimizations and their impact on the performance for subsequent research and development in this field. We will hence be focusing on optimizing our building blocks as a priority.

6.2 Design Algorithm

As shown in the software training section, we have a symmetric architecture for our network which alleviates many of the design challenges presented by deep networks especially when it comes to implementation on hardware platforms. Moreover, it provides design flexibility and can include more number of layers keeping in view the hardware constraints.

Looking at the architecture, we also deduce the following points.

1) First layer is designed to handle input image of size 32x32 with three channels for red, blue and green. This effectively reduces the requirements of the filter size to depth three for the first layer.
2) Input data width for first layer is 8 bits to deal with the pixel values ranging from 0-255.
3) Subsequent convolution layers deal with greater number of input feature maps. The data width however, is restricted to a single bit and we will implement the required optimizations allowed through the implementation of single bit wide convolution.

4) We have designed a normalization layer which is embedded in each of the convolution blocks. This layer, as explained in previous section, becomes important in any reduced precision network to enable faster convergence during the training process.
5) Pooling layers periodically reduce the feature map dimension in the network as we go deeper. This allow us to limit the growing amount of data as we increase the filter numbers in subsequent convolution layers.
6) In each of the convolution layers, the filter size is fixed to 3x3 which require a 0 padding of two for all the layers. Hence we keep the feature map size constant until we want to reduce it through a pooling layer. This controller reduction in size also makes hardware design much simpler and easy to manage.
7) Each level of the network has a binarization layer which acts as the activation function and converts the data back to one bit. Theoretical principles of all these layers have already been explained in the preceding sections. We visualize design in figure 34.

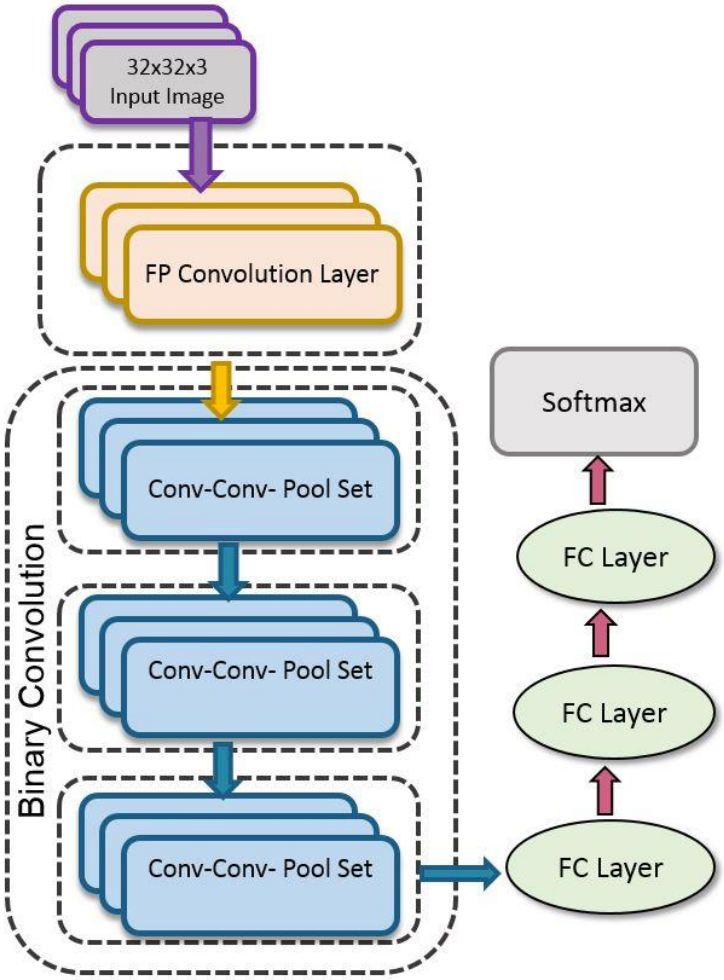


Figure 34: Network Architecture to be implemented on FPGA.

6.3 Hardware Convolution Techniques

Since convolution occupies most of the processing time in the network, we discuss a few convolution techniques that have been adopted on hardware which offer a potential solution for us. These techniques can also be extended to the fully connected layers which can be thought of as a simpler version of convolution. We adopt the below mentioned techniques throughout our design based on the layer requirements.

6.3.1 Matrix-Matrix Multiplication

One approach for convolution is to convert it into a matrix-matrix multiplication. This would be more like an array multiplication from a programming perspective. It's a relatively simple approach which requires filter data and image data to be fetched into a single array format and then multiply and add the data. In terms of a binary weight and image scenario, this seems even more beneficial since all the operations are XNOR and bit count (section 3.5.1) and not even multiplications and additions. Hence performing matrix manipulations rather than pure convolution makes even more sense. One drawback of this approach is that data placement in memory needs to be in a specified format for efficiently performing the matrix-matrix multiplication. Also, since we are reading portions of the image and filters at a given time it is harder to keep track of the convolution process. We explain the process using figure 35 where matrix A and B are represented in memory as 2 arrays. This array representation is the formatting that is done as part of data preprocessing or offline processing. To generate O1 of output matrix O, we XNOR-bitcount the appropriate indices as shown in figure. Similar process is repeated to generate O2-O9. The output is again represented in memory as an array and is illustrated as a matrix in figure 35. Tradeoff for this simplicity is the added code complexity for correct index access of the matrices.

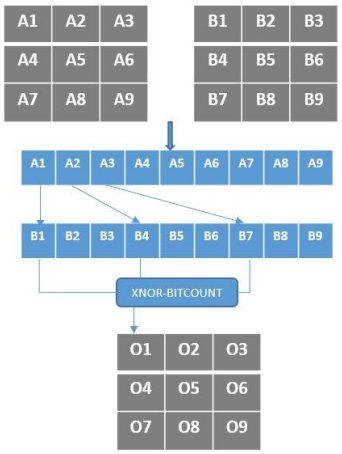


Figure 35: Illustration of matrix multiplication using preformatting and indexing

As mentioned above, the technique requires data preprocessing/formatting in a specific order in memory. This is also called offline processing since it is done before running the network. Offline data processing time is not considered during analysis of CNN architecture performance.

6.3.2 Line Buffer and Sliding Window

The most popular approach for full precision convolution on hardware platforms is using a line buffer and a sliding window to mimic the exact behavior of the convolution process. Many of the FPGA vendors even provide line buffer and windowing functions as part of their standard library due to their wide use. Using this approach requires minimal data preprocessing and the 2D image can be stored as such in memory. Figure 36 shows a schematic of sliding window.

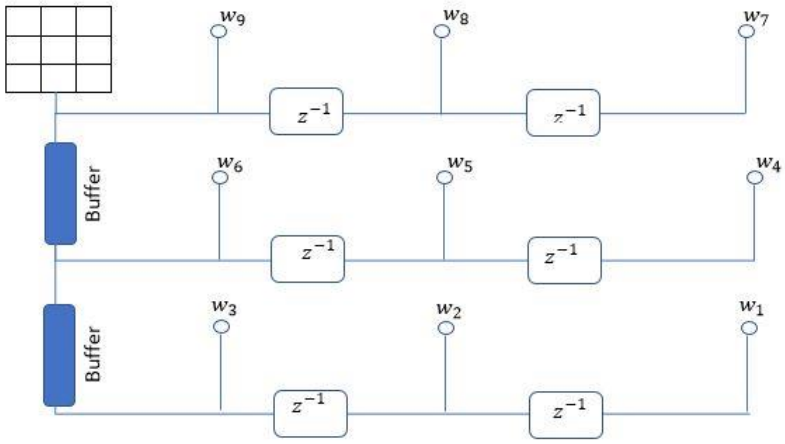


Figure 36: Sliding Window Schematic.

In figure 36, we see 2 line buffers which are a first in first out structure. As a design principle for correct functionality of the sliding window, *(filter Dim – 1)* number of line buffers are present in the design. In case of using a filter of dimension *3x3*, we would require *3-1=2* line buffers in the window. Each line buffer will have a depth equal to the width of the input feature map or image size minus 1. We also have additional delays represented as z^{-1} and can be implemented by a simple register. These delay lines place the appropriate weight and pixel together for multiplication. Weights of filter are represented by *w1*-w9. To better understand the working of the sliding window, let us take the example of using a 3x3 filter over a 32x32 image. Image pixels are read in and filled in buffers before start of processing. When pixel 1 reaches *w1*, we have pixels 2 and 3 at *w2* and *w3* respectively. Remaining pixels of the row are stored in buffer 1. Pixels 33, 34 and 35 are at weight lines *w4*, *w5* and *w6* respectively. Again, the remaining pixels of the row are store in buffer 2 and pixels 64, 65 and 66 are at weights *w7*, *w8* and *w9*. When more pixels are read in, we get a sliding window effect. Hence, such a structure can imitate a sliding window effect

in a convolution process. This is the closest possible approach to an actual theoretical convolution implementation on hardware.

Another advantage of such an approach is the pipelined nature of the architecture. In hardware platforms such as FPGAs we try and pipeline our design as much as possible to keep all the available resources busy. This maximizes throughput even if the latency is large.

The drawback of such an approach is the complexity when we operate on high dimensional input data. For high dimensions of input feature maps, such as 128 or greater, the design not only becomes very complex but also utilizes large amounts of resources. Fortunately, for a binarized convolutional network, the first layer has only three layers i.e. R,G and B and hence seems to be suitable for such an implementation.

6.4 Scheduled Hardware Pipeline

One of the biggest advantages of using binarized networks, as mentioned before, is the exceedingly small memory size. This enables all the required weights to be packed on the chip memory in block RAM. It also has a 1-bit wide output feature map from each convolution layer and hence each layer output can be stored in block RAM. We can utilize this for our advantage and pipeline the architecture. We aim to design such an architecture that each layer has its weights store on BRAM and output stored in another on-chip BRAM. This output RAM is then accessed by the next layer after the previous layer has completed its operation. Hence, we maximize our throughput by utilizing the on chip memory and scheduling layer operations by a controller in a pipelined fashion. This schedule operation is done by a scheduler which continuously monitors the state of each layer and controls the start point of each layer. The scheduler can be implemented in the form of a state machine on the programmable logic itself. However, owing to the nature of our hardware platform, which has a Zynq SOC architecture, we can utilize the on board ARM core or the PL based Microblaze for this operation. We visualize the pipeline in figure 37.

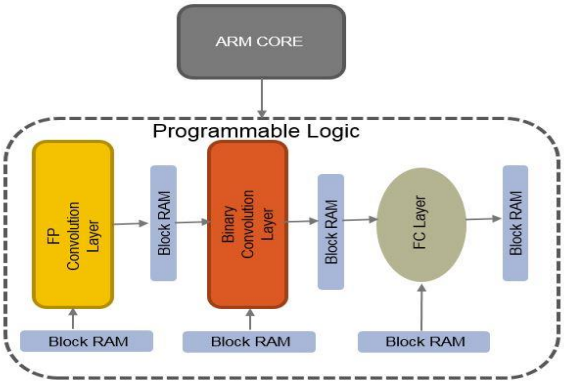


Figure 37: Overview of Network Architecture with Zynq Core

CHAPTER 7

HARDWARE IMPLEMENTATION

We now begin to explain our implementation of the binary CNN on FPGA. We have already discussed the nature of our network and the HLS tool required for the implementation. As per the division in our network, we discuss the layers in order of their presence in the network.

7.1 First Convolutional Layer (C-1)

The first layer we design will be the convolution layer C-1. We have the following requirements from this layer

1) Input is an image of size 32x32 with three layers for red, green and blue.
2) The input pixel data width is 8 bit to represent values from 0-255 which cover the entire range of the spectrum.
3) Performs convolution of 1 bit weighted filters.
4) Filter dimensions are 3x3 with depth of 3 for the three layers of the inputs.
5) A total of 64 filter comprise the convolution layer
6) Each filter produces a 32x32x1 output feature map through convolution
7) Total output size from 64 filters is 32x32x64
8) Filter weights are stored in BRAM
9) Input image is stored in DDR off chip.
10) Output feature map has data width of 1 bit wide
11) Output feature map is stored on BRAM
12) The layer generates intermediate real valued outputs
13) The layer has normalization function which is applied to the intermediate outputs from the layer.
14) The convolution layer has binarization capability which reduces the real valued outputs to binary weighted outputs
15) The final output feature map is binary weighted

Now that we are aware of the requirements and functionality of the first convolution layer, we can start designing the layer. We will discuss the approach taken for the design of the layer, coding style, optimizations applied and the final design selected.

7.2 Single Data Stream

First design created was a single data stream design. We refer to figure 38 and explain the operation of this architecture. One of the three layers (R, G, B) of the input image is transferred from DDR to the convolution layer on programmable logic. Filter weight values are fetched from

the BRAM one by one and stored in the *w0-w9* as were shown in our illustration of the sliding window (figure 36). Input image layer is streamed in the convolution block and multiplied with the weight values to get the convolution result. This result is the intermediate output of the layer. The intermediate output is stored in a block RAM. After this, the second layer of the R, G, B image is streamed in and the intermediate result is saved in the block RAM. This operation is repeated for the third layer as well. Once the convolution operation is applied to all three layers of the image, the result is summed using a scalar addition of the image maps. This gives the total convolution result. Next step is the pixel by pixel multiplication of the summed output with normalization parameters in the normalization block. The final step in the process is the binarization of each pixel using the binarization function detailed previously (section 3.5.3). Final result from the binarization block is stored in another block RAM which will subsequently be used as an input for the next layer. We show this architecture in figure 38. Algorithm is summarized in section 7.2.1

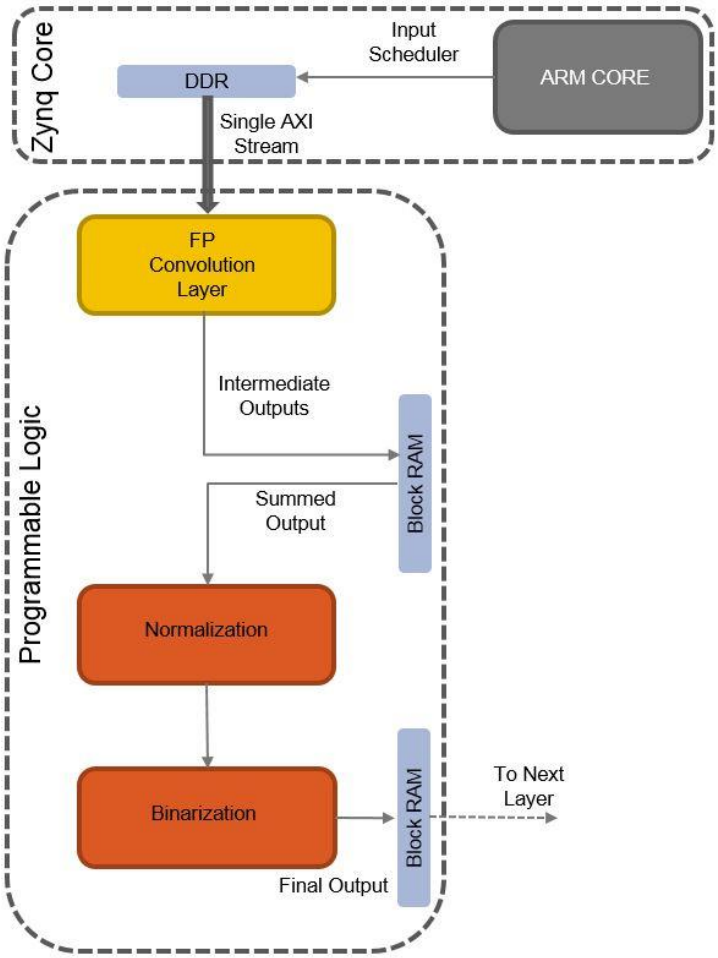


Figure 38: Single Data Stream Design

7.2.1 Algorithm (single stream)

Function firstlayer (single input stream, final output, stream number, filter weights, normalization parameters)

While $iterations\ (i)\ \leftarrow 1\ to\ 3, do$

While $pixelcount\ \leftarrow 1\ to\ totalpixels, do$

Read in pixel value

Insert it in Line Buffer

For $width\ \leftarrow 1\ to\ filterwidth, do$ *(slides window)*

Pipeline the design (HLS PRAGMA)

For $height\ \leftarrow 1\ to\ filterheigth, do$ *(slides window)*

For $filters\ \leftarrow 1\ to\ totalfilters, do$

Read Weights from memory

Push filter weight in window

If weight==0

Pixelval=~Pixelval

Else

Pixelval=Pixelval

Sum (Pixelval)

Push in buffers

Sum (all windows)

Push values in temp Bram

Add Result (i-1) to Result (i)

Push back in temp Bram

Normalize temp Bram data

Binarize temp Bram data

$Bram\ (output\ map) \leftarrow Binarized\ data$

7.2.2 Results

Table 14: Single Stream Resource Requirement

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	6867
FIFO	-	-	-	-
Instance	0	-	38	248
Memory	9	-	48	12
Multiplexer	-	-	-	11227
Register	-	-	4052	-
Total	9	0	4138	18354
Available	1824	2520	548160	274080

We see the resource utilization of the design in table 14. We look at the block RAM consumption, flop flops used as well as the look up tables (LUTs). The design consumes 9 out of 1824 BRAM blocks (this number is for design consumption, filter weights would require extra 1.7Kbits of BRAM). All the BRAM consumption goes in to memory as indicated in table 14. This memory stores the intermediate results generated. 4138 out of 548160 FFs are used by the design. Most of them are used up in registers. 18354 of 274080 LUTs are used. Most of the LUTs are used in multiplexers which implement the binary multiplication operation in this design. Looking at the resource constraints of the board and the resource utilization of the layer, we can see that with this performance we can fit in at least 6 convolution layers on the chip (we are consuming less than 10% of the available resources). We deduce that this design satisfies the resource constraints of the FPGA board. We also look at the performance of the designed architecture. We run the performance analysis tool which gives us the total latency of the block as 461000 clock cycles for a single pass of the design. We recall that our design operates three times on the three input layers of the image. We calculate that the three passes have at least thrice the time consumption of a single pass and hence summarize the performance of the design to process the complete image in table 15.

We now try and improve performance of this layer as we would like to get maximum possible throughput from our design without increasing the resource consumption.

Table 15: Latency for Single Stream Design

Iterations	Latency (cycles)
Single Pass	461000
Three Passes	~1.5 million

7.3 Multiple Data Stream

We identify that the single stream overhead can be avoided if we transfer three streams in parallel through the three high performance ports of the Zynq Core. It is difficult to predict the resource consumption of such algorithms until we have a baseline and hence keep the earlier design as a fall back to a simpler algorithm in case this is too resource intensive. The design flow is simpler in this with the Zynq core transferring the three layers at the same time. We replicate our design in three copies to manage the three layers at the same time and add the result at the end to get a final output. We illustrate the architecture in figure 39. This is similar to the previous design. Except we are now transferring all three layers of the input image to the convolution block simultaneously. Intermediate outputs are again stored in a block RAM. These outputs are then summed together, normalized and binarized as was done in the single data stream design. The final output is stored in a block RAM to act as input to next layer. Design is illustrated in figure 39 and summarized in section 7.3.1

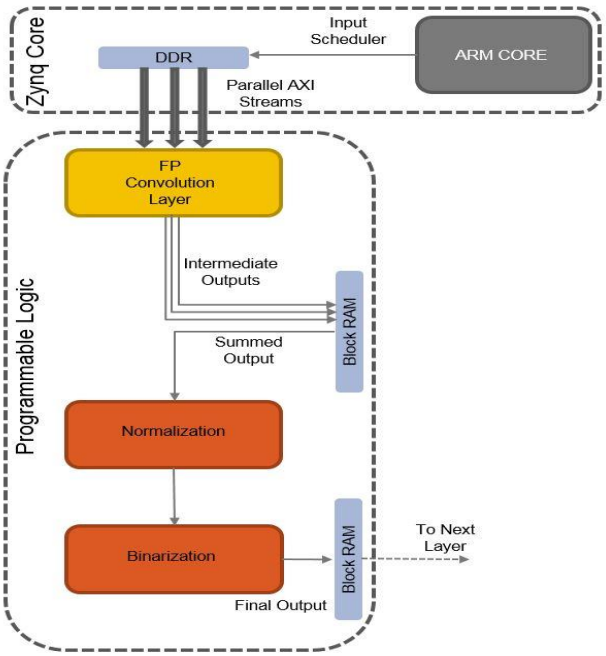


Figure 39: Multiple Data Stream Design

7.3.1 Algorithm (Multiple Stream)

Function firstlayer (stream1, stream2, stream3, final output, filter weights, normalization parameters)

While $pixelcount \leftarrow 1\ to\ totalpixels, do$

Read in pixel value from each stream

Insert it in Line Buffer

For $width \leftarrow 1\ to\ filterwidth, do$ *(slides window)*

For $height \leftarrow 1\ to\ filterheigth, do$ *(slides window)*

Pipeline the design (HLS PRAGMA)

For $filters \leftarrow 1\ to\ totalfilters, do$

Read Weights from memory

Push filter weight in window

If weight==0

Pixelval=~Pixelval

Else

Pixelval=Pixelval

Sum (Pixelval)

Push in buffers

Sum (all windows)

Push values in temp Bram

$temp\ Bram \leftarrow Sum(temp\ (1) + temp(2) + temp(3)$

Normalize temp Bram data

Binarize temp Bram data

$Bram\ (output\ map) \leftarrow Binarized\ data$

7.3.2 Results

We do a design synthesis again and get the results for resource consumption of the design in table 16. We look at the block RAM consumption, flop flops used as well as the look up tables (LUTs).

Table 16: MultiStream Design Resource Utilization

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	60976
FIFO	-	-	-	-
Instance	0	-	1313	3706
Memory	331	-	144	36
Multiplexer	-	-	-	29677
Register	-	-	4649	-
Total	331	0	6106	94395
Available	1824	2520	548160	274080
Utilization (%)	18	0	1	34

We see in table 16 that the resource consumption has drastically increased. With 18% design BRAM consumption and adding the filter weight data to it, we will quickly run out of memory especially when we reach the fully connected layers which have much more memory requirements. We again see that all the BRAM consumption goes into memory generation (331 blocks). We have a 50% increase in FF consumption but it is still only 1% of the available FF resources. The LUT consumption has exceeded much more sharply, consuming 34% of the available chip resources to model different expressions and multiplexers for binary multiplication operations. We cannot fit the required number of convolutional layers at this rate of resource consumption. Hence we can see that if we reduce latency, we cannot get the appropriate size of our design. We rethink our approach to the problem and reduce resource consumption without impacting latency.

7.4 Design Optimization

We can see that we are facing three problems with our design so far. These are memory consumption, LUT consumption and high latency. We will perform two optimizations to address these issues and see the performance of our design.

7.4.1 Temporary Memory Removal

It can be identified through looking at the flow diagram of the previous design that a temporary block RAM has been added in the design. It can be observed that if we directly add the results from the three streams, there would be no need for the extra BRAM. This would require some extra on chip LUT consumption which we will try to compensate in the next section.

7.4.2 LUT Consumption

In the previous designs, in order to get minimum latency from our design, the design was pipelined and unrolled for performance. While a parallel design performed better than a single stream design, it increased the LUT consumption to intolerable levels. Hence we need to reroll the design and sacrifice on performance to save on the valuable on chip resources. Hence we reduce the unrolling factor in our design and sacrifice the performance. We will see in the next section, the optimization we perform to recover on this performance loss.

7.4.3 Filter Weight Packing

So far we have been reading in filter weights from the BRAM one at a time. With even the implementation of a dual port BRAM, we can only reduce the memory access over head so much. Each time we read a memory location, we add latency to our design. This is because memory access is slow and is limited by the memory access bandwidth. This might be less for on chip BRAM then external DDR access, but it is still there. However, we realize that the filter weights are only 1 bit wide and not 32 bit like a full precision network. With Zynq architecture, we can access 1-1024 bits at a time. Hence we formulate the following scheme.

1) We pack filter weights together.
2) Each filter has 3x3x3=27 weights for the three input layers
3) We have a total of 64 filters.

Hence we can pack the 27 weights of each filter into a single 32 bit location of the Bram. We will waste 15% of the memory for ease of design and pack the remaining 5 bits of the 32 bit integer with zeros. Instead of doing 27x the number of filters, we only access the memory “no. of filter” times and hence drastically reduce memory read overhead. In the convolution block, we unpack the read filter data, separate and store it into internal LUTs. This does not have a substantial resource overhead since storing 27 bits of data at a given time is a minor cost compared to the gains we can potentially get. One drawback of this approach is the memory packing overhead. However, this can be done offline before the operation of the layer and hence does not add to the processing time of the design. We now formulate the new algorithm for this approach. We summarize the optimizations as follows. We have removed the temporary memory to store three layers of intermediate outputs. Instead, we sum the outputs of the three layers together before storing. In this way, only a single layer of intermediate outputs needs to be stored. We also reduce the unroll factor in our design through the use of pragmas. Finally, we make use of efficient filter weight packing in a single location to reduce memory access by a factor of 32. This has been summarized in the next section.

7.4.4 Algorithm (Optimized Design)

Function firstlayer (stream1, stream2, stream3, final output, filter weights, normalization parameters)

Replicate each instruction for three streams

While $pixelcount \leftarrow 1\ to\ totalpixels, do$

Read in pixel value from each stream

Insert it in Line Buffer

Read Weights from memory *(read weights outside the loop*

For $width \leftarrow 1\ to\ filterwidth, do$ *(slides window)*

For $height \leftarrow 1\ to\ filterheigth, do$ *(slides window)*

For $filters \leftarrow 1\ to\ totalfilters, do$

$weight \leftarrow Local\ array\ [index]$

Push filter weight in window

If weight==0

Pixelval=~Pixelval

Else

Pixelval=Pixelval

Sum (Pixelval)

Push in buffers

Sum (all windows)

$temp\ Bram\ (index) \leftarrow Sum(\ windows)$

Normalize temp Bram data

Binarize temp Bram data

$Bram\ (output\ map) \leftarrow Binarized\ data$

7.4.5 Results

We synthesize the new design again and look at the resource consumption in table 17.

Table 17: Optimized Design Resource Utilization

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8380
FIFO	-	-	-	-
Instance	0	64	3843	5761
Memory	69	-	126	51
Multiplexer	-	-	-	4958
Register	-	-	4457	-
Total	69	64	8426	19150
Available	1824	2520	548160	274080
Utilization (%)	3	2	1	6

From table 17, we see the resource consumption of the optimized design. BRAM consumption has decreased from 18% to 3% of available resources. This is sufficient reduction to implement 5 more convolution layers of similar BRAM consumption. FF consumption has slightly increased but it is still ~1% of available resources and hence should not be a problem area. LUTs had the highest consumption level in our un-optimized design. We see a reduction in LUT consumption from 34% to 6%. At same level of LUT consumption, this should prove to be sufficient to implement 5 more convolutional layers.

We do a performance analysis and compare it with our previous results.

Design	**Latency (cycles)**
Un-optimized Single Stream	1.5 million
Un-optimized Parallel Stream	460000
Optimized	510000

The design requires 510000 clock cycles to process the input image. We see that we have achieved performance close to parallel input stream design with resource utilization matching that with a single stream design.

7.5 Binary Convolution Layers (C-2 to C-6)

The next part of the network is the binary convolution layer. After the first convolution layer, our feature maps have been binarized. Hence we can apply the binary convolution to each of the subsequent feature maps. We finalize the design requirements of these layers as

1) Input is a feature map. This can be of dimensions 32x32, 16x16 and 8x8. Their numbers depend on the number of filters in the previous layer and can range from 64 to 128.
2) The input pixel data width is 1 bit to represent values of either 1 with a 1 or -1 with a 0.
3) Performs convolution of 1 bit weighted filters.
4) Filter dimensions are 3x3 with depth of 64 or 128 depending on the number of output feature maps from previous layer.
5) Each filter produces a 32x32, 16x16 or 8x8 output feature map through convolution and pooling
6) In case pooling layer is not present, output dimensions are same as input dimensions
7) In case of presence of pooling layer, the map size is halved as explained in the pooling layer operation theory in previous sections.
8) Filter weights are stored in Bram
9) Input feature map is read from Bram
10) Output feature map has data width of 1 bit wide
11) Output feature map is stored on Bram
12) The layer generates intermediate real valued outputs
13) The layer has normalization function which applies training parameters on the intermediate outputs from the layer.
14) The convolution layer has binarization capability which reduces the real valued outputs to binary weighted outputs
15) The final output feature map is binary weighted

We now design the binary convolution layer. We design the first layer and apply optimizations to it until we can get satisfactory performance from it. After we have finalized the design, we can change the filter number and sizes and get the rest of the layers.

7.5.1 Unoptimized Algorithm

At the binary convolution layer, we can create a true binary convolutional algorithm. As mentioned in previous sections, this includes XNOR followed by bit count operation. Also, at this stage, since the input feature map is now stored on the block RAM, there is no data streaming operation required from the Zynq core. The flow map in figure 40 shows a basic binary convolution layer. The pooling layer might or might not be present in all layers depending upon their position in the network.

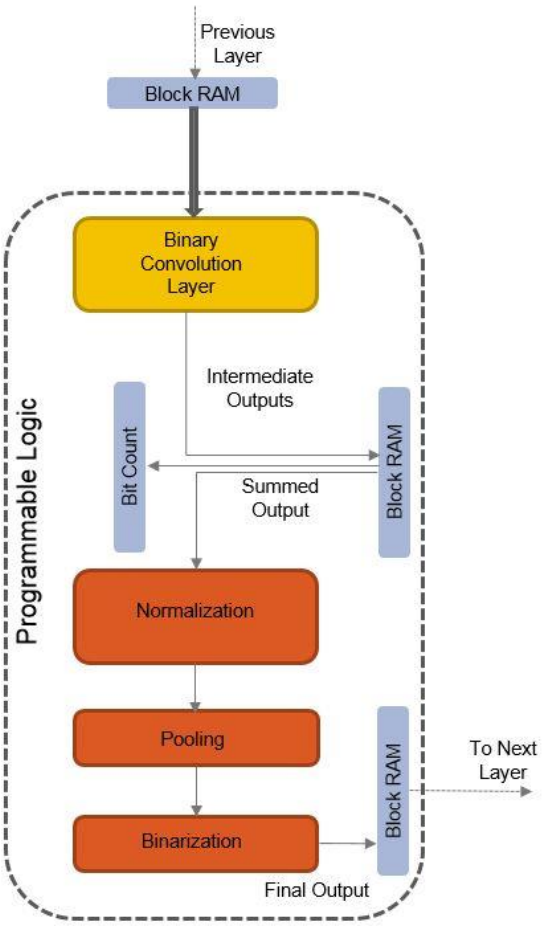


Figure 40: Unoptimized Binary Convolution Layer

The binary convolution layer reads in filter weights from block RAM and performs the XNOR operation for each filter at a given position. It stores the intermediate output values in block RAM. These stored values are the result of XNOR operation between input map and the filter weights. After this, it sends it to the bit count block which performs the bit counting operation on the values. This is the final result of the convolution operation. The result is again stored in the block RAM. This is then sent to normalization layer which multiplies a normalization factor with each of the pixels and generates a normalized output. If a pooling layer is present in the layer, a pooling operation is performed to reduce the size of the output maps. Binarization operation is performed at the end after normalization and pooling steps. Final binarized output is stored in the block RAM which would act as an input to the next layer. This design is illustrated in figure 40. We formulate the algorithm in section 7.5.2.

7.5.2 Algorithm (Un-Optimized Design)

Function binarylayer (inputfeaturemap, final output, filter weights, normalization parameters)

While $pixelcount \leftarrow 1\ to\ totalpixels, do$

Read in pixel value from Block Ram

Store value on PL

For $width \leftarrow 1\ to\ filterwidth, do$ *(slides window)*

$For\ height \leftarrow 1\ to\ filterheigth, do$ *(slides window)*

$For\ depth \leftarrow 1\ to\ filterdepth, do$

#pipeline loop

$For\ filters \leftarrow 1\ to\ totalfilters, do$

$weight \leftarrow Block\ Ram$

$featuremap \leftarrow BlockRam$

XNOR (weight,featuremap)

Push value in buffer

$temp\ Bram \leftarrow\ bit\ count\ (buffer[all\ filters])$

Normalize temp Bram data

Pool temp Bram

Binarize temp Bram data

$Bram\ (output\ map) \leftarrow Binarized\ data$

We analyze the performance of the algorithm in the next section

7.5.2 Results

Table 18: Unoptimized Design Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	5461
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	3	-	16	8
Multiplexer	-	-	-	1416
Register	-	-	927	-
Total	3	0	979	6925
Available	1824	2520	548160	274080

Design	Latency (cycles)
Binary Conv (XNOR-Bitcount)	>90 million

From table 18, we see that we have only used 3 blocks of BRAM in the design which is 0.1% consumption. Similarly, both FF and LUT consumption numbers are very small (less than 0.1% resource utilization). While the resource utilization is good as was expected from the usage of binarized weights, the latency has inflated to a very large number of 90 million clock cycles. One reason for this is the inability to completely pipeline the design since the synthesis tool fails to generate a pipelined design for the XNOR and bit count operations.

Another reason for this is the repeated use of the bit count operation which can slow down the entire pipeline quite a bit considering the large number of bits to be counted as well as repeated use of the block for each pixel and each filter. Hence we will also optimize this. One more problem area that we can identify here is the large number of weight values. Even though the weights are only 1 bit wide, we still have to access the memory each time the weight is read on to the programmable logic for convolution. For very deep filters such as 64 and 128 in the subsequent layers, the number of weights increases drastically. This problem will be even more amplified when we move on to the fully connected layers which have substantially higher number of weights than the convolution layers. We will now see how we optimize the above stated issues in the next algorithm.

7.5.3 Optimized Algorithm

To remedy the above mentioned problems, we apply the following optimizations in our algorithm

1) We reduce the number of memory accesses in a similar way we did in the first convolution layer. We pack the filter weights in larger data types. These are then read and split up on chip. This saves memory access time
2) We eliminate the bit count function and instead implement a look up table based approach. Since one layer of filter has 9 weights, we can implement a look up table of depth 512 and add all possible results of bit count in the look up table for the 9 bits. Hence we don't have to calculate the bit count operation repeatedly
3) An added advantage of the look up table based approach and the elimination of bit count function is that now we can implement pipeline in the design at a higher level without consuming too much resources.

We implement the optimizations and look at the results.

7.5.4 Results

Convolution Layer 2

Table 19: Layer 2 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	12640
FIFO	-	-	-	-
Instance	0	64	2613	4491
Memory	79	-	32	16
Multiplexer	-	-	-	2037
Register	-	-	2306	-
Total	79	64	4951	19184
Available	1824	2520	548160	274080
Utilization (%)	4	2	~0	6

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer2	64	32x32	64	16x16	4.6 million

Table 19 summarizes convolutional layer 2 which is the first binarized convolutional layer and has an input dimension of 64 with each input map of size 32x32. The layer applies 64 filters on the inputs to generate 64 dimensional output. It also applies a pooling layer to reduce the output size to 16x16. In terms of performance, we see from table 19 that the design is much slower than the first convolutional layer since it has a much higher number of filters. The convolution block requires 4.6 million clock cycles to complete the operation. For resource utilization, we have a 4% utilization in block RAM and a 6% utilization of LUTs. We have so far consumed 12% LUT resources in the network pipeline.

Convolution Layer 3

Table 20: Layer 3 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	24256
FIFO	-	-	-	-
Instance	0	128	4332	7147
Memory	38	-	0	0
Multiplexer	-	-	-	3571
Register	-	-	3795	-
Total	38	128	8127	34974
Available	1824	2520	548160	274080
Utilization (%)	2	5	1	12

Layer	**Inputs**	**Input Size**	**Outputs**	**Output Size**	**Latency (cycles)**
Layer3	64	16x16	128	16x16	2.2 million

Table 20 summarizes convolutional layer 3 which is the next binarized convolutional layer and has an input dimension of 64 with each input map of size 16x16. The layer applies 128 filters on the inputs to generate 128 dimensional output. It does not apply pooling and the output size is maintained at 16x16. In terms of performance, we see from table 20 that the design takes 2.2 million clock cycles to run. This is faster than the previous convolutional layer due to a smaller input size of 16x16. For resource utilization, we have a 2% utilization in block RAM and a 12% utilization of LUTs. We have so far consumed 18% LUT resources in the network pipeline.

Convolution Layer 4

Table 21: Layer 4 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	24306
FIFO	-	-	-	-
Instance	0	128	4539	7630
Memory	43	-	0	0
Multiplexer	-	-	-	3627
Register	-	-	3838	-
Total	43	128	8377	35563
Available	1824	2520	548160	274080
Utilization (%)	2	5	1	12

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer4	128	16x16	128	8x8	4.4 million

Table 21 summarizes convolutional layer 4 which has input dimension of 128 with each input map of size 16x16. The layer applies 128 filters on the inputs to generate 128 dimensional output. It also applies a pooling layer to reduce the output size to 8x8. In terms of performance, we see from table 21 that the design takes 4.4 million clock cycles to run. This is slower than the previous layer due to more filter depth of 128. For resource utilization, we have a 2% utilization in block RAM and a 12% utilization of LUTs. We have so far consumed 30% LUT resources in the network pipeline.

Convolution Layer 5

Table 22 summarizes convolutional layer 5 which has input dimension of 128 with each input map of size 8x8. The layer applies 128 filters on the inputs to generate 128 dimensional output. It does not apply pooling and the output size is maintained at 8x8. In terms of performance, we see from table 22 that the design takes 1.1 million clock cycles to run. This is faster than the previous layer due to smaller input image size of 8x8. For resource utilization, we have a very small block RAM consumption of 13 blocks and a 12% utilization of LUTs. We have so far consumed 42% LUT resources in the network pipeline.

Table 22: Layer 5 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	24265
FIFO	-	-	-	-
Instance	0	128	3736	7013
Memory	13	-	0	0
Multiplexer	-	-	-	3571
Register	-	-	3791	-
Total	13	128	7527	34849
Available	1824	2520	548160	274080
Utilization (%)	~0	5	1	12

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer5	128	8x8	128	8x8	1.1 million

Convolution Layer 6

Table 23: Layer 6 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	24265
FIFO	-	-	-	-
Instance	0	128	3901	7389
Memory	15	-	0	0
Multiplexer	-	-	-	3627
Register	-	-	3794	-
Total	15	128	7695	35281
Available	1824	2520	548160	274080
Utilization (%)	~0	5	1	12

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer6	128	8x8	128	4x4	1.1 million

Table 23 summarizes convolutional layer 6 which is the last convolutional layer in the pipeline and has an input dimension of 128 with each input map of size 8x8. The layer applies 128 filters on the inputs to generate 128 dimensional output. It also applies a pooling layer to reduce the

output size to 4x4. In terms of performance, we see from table 23 that the design takes 1.1 million clock cycles to run. For resource utilization, we have a very small block RAM consumption of 15 blocks and a 12% utilization of LUTs. We have so far consumed 54% LUT resources in the network pipeline.

After this we design the fully connected layer. The output from convolutional layer 6 is stored in block RAM and is taken as input to the fully connected layer. Our total resource utilization so far is 54% LUT consumption. Block RAM and FF consumption is under 10% which leaves space to add more layers as well.

7.6 Fully Connected Layer

The last layer in the convolutional neural network is the fully connected layer. In our network, we have three fully connected layers on the programmable logic. We formulate the design requirements of the fully connected layer.

1) Input is a single dimensional array which represent the values coming into the fc layer. This input is equal to the number of output pixels from the previous layer.
2) The first layer will have 2048 input pixels coming in. These values are also binary weighted and hence all fc layers only deal with binary valued inputs.
3) All weight values are binary weighted
4) Number of weight values depend on the number of input and number of outputs
5) If number of inputs are [x,1] and output dimension is [1,y], then the dimension of weight matrix is [x,y]
6) Weights and inputs are multiplied and accumulated in a similar way as done in the convolution layer using XNOR and bit count operation
7) The result of this operation is real valued
8) The real valued intermittent values are normalized in the FC layer
9) Normalized values are binarized before sending it to the next FC layer.
10) Values from last layer are not binarized. Softmax function is applied to outputs of the final layer for conversion to class probability scores

Figure 42 shows operation of FC layer. We use the experience from the previous layers. We use an LUT based approach for bit count operation. We use weight packing for efficient memory fetch operations. We unroll the design for performance.

From figure 41, we see a pipelined approach to the design of the fully connected layers as well. Output from last binary convolution layer is taken as input to the fully connected layers. FC1 takes input and applies a matrix-matrix multiplication to generate the output. For our design, Conv6 generates an output of 4x4x128. Input of FC1 has 2048 nodes and output side has 1024 nodes. In a fully connected layer architecture each node of input is connected to each node of output. This

means that we will perform a matrix-matrix multiplication of *1 x 2048* matrix with *1024 x 1024* matrix. The matrix with 1 x 2048 size contains the inputs coming from the convolution layer. 1024 x 1024 matrix contains the weights which will be multiplied with the inputs to generate 1 x 1024 outputs. The multiplication operation in this layer is also an XNOR and bit count operation. We apply a look up table based bit count approach as was done in previous layers. Intermediate values of outputs are real valued. We multiply each output with normalization parameters. Binarization function finally converts each output to a binary value.

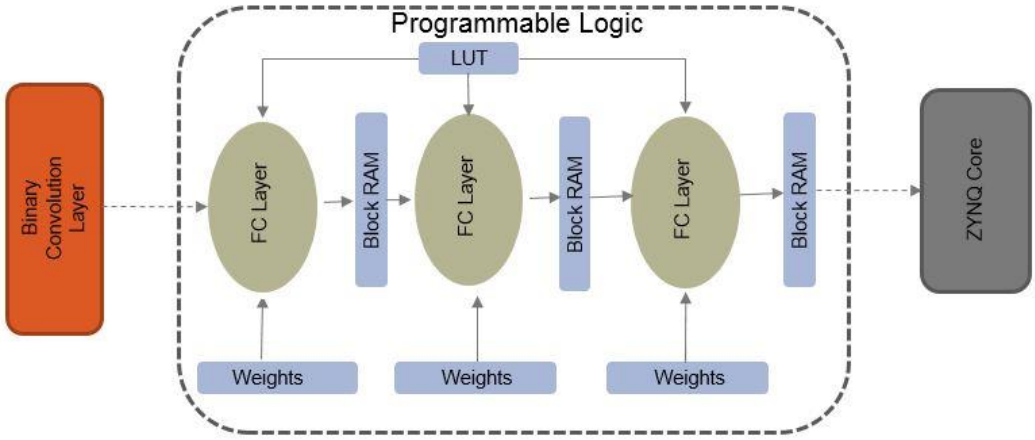


Figure 41: Fully Connected Layer Architecture

7.6.1 Algorithm

As mentioned in the previous sections, the fully connected layer has the bigger percentage of the neural network weight values. This can potentially cause performance issues due to limited bandwidth of memory fetch. Hence we aim to reduce the number of memory accesses to avoid this bottleneck. To achieve this, we perform the following optimizations

1) Pack single bit weight and input values into a 32 bit data type. This reduces the number of memory fetch operations.
2) For the matrix-matrix multiplication, we would need to read in the input and the weight values multiple times to complete the operation. We can avoid this if we read the input values once, compute the temporary results for each column of the weight matrix and store in a buffer until all the input values have been fetched. In this way we can greatly reduce the number of read operations and increase performance. To elaborate this further, we look at figure 42. Suppose we are multiplying a matrix with dimensions 1x8 with a matrix of dimensions 8x3. We should get an output of order 1x3. In order to perform the multiplication, we read in first element of row A and multiply with first element of col B1.

Similarly we read in remaining elements of row A and col B1 to calculate the total sum which would be the first entry of the final output. This process is repeated for all outputs. Since we are dealing with binary values of inputs and weights, we perform the optimization. We read in block '*a'* which are the first 4 entries of row A. We read in first 4 entries of col B1 which are *b1,* multiply 'a' and 'b1' and store result. We similarly read in *b2* and *b3,* multiply with 'a' and store intermediate results. Once all calculations using '*a'* are complete, we fetch in the next block *a'* (a prime) and repeat the process with *b1', b2'* and *b3'.* This optimized memory fetch approach greatly reduces the number of memory accesses. Figure 42 illustrates this block based approach.

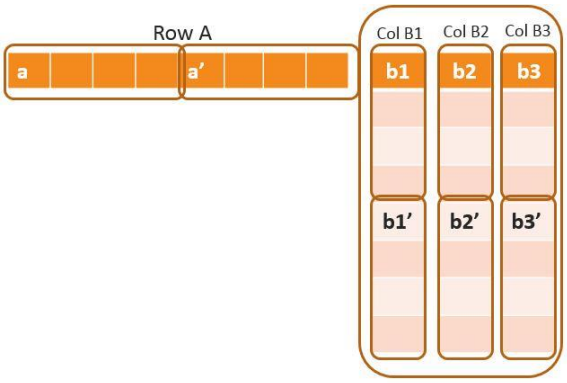


Figure 42: Illustration of efficient memory fetch scheme in FC layer

Section 7.6.2 presents pseudocode for this algorithm.

7.6.2 Algorithm (FC Layer)

Pseudocode

```
Depth= (number of inputs/datatype)  //this packs bit values into data type
Outfc=number of outputs


for (int fil=0;fil<depth;fil++)  //initialize outer loop
{
indata=featuremap[fil];                 //read in first 16 bits (we consider data type as short)
for (int outs=0;outs<outfc; outs++)    //inner loop
{
filread=weights[fil][outs];             //read in 16 bits of 'outs' column
for (int buf=0;buf<16;buf++)            //loop for XNOR
{
#pragma HLS unroll                      //unroll loop
buffer[buf]=(indata[buf]==filread[buf]);       //perform xnor
}
```

```
sum[outs]+=lut[buffer];                    //read in xnor bit count result from LUT and store
result in temporary buffer for 'outs' column
}
}
```

We use this to synthesize the last three layers of our network which are the fully connected layers and analyze performance and resource consumption

7.6.3 Results

Fully Connected Layer 1

Table 24: FC1 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	440
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	33	-	0	0
Multiplexer	-	-	-	107
Register	-	-	108	-
Total	33	1	144	587
Available	1824	2520	548160	274080
Utilization (%)	1	~0	~0	~0

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer 1	2048	2048x1	1024	1024x1	395651

Fully connected layer FC1 has input dimension of 1 x 2048 and an output dimension of 1 x 1024. This implies that it stores 2048 x 1024 weight values for the matrix multiplication. In terms of performance, we see from table 24 that the design takes 395651clock cycles to run. We had already discussed how most of the computations of the architecture are pooled in the Conv layers. We see from the analysis in table 24 that FC1 has lowest latency so far due to the reduced computation load. For resource utilization, we have a very small block RAM consumption of 33 blocks and a negligible utilization of LUTs and FFs. However, the RAM consumption is for the layer design and it does not take into consideration the network weights which are of the order

2048 x 1024. Hence the network has a large memory and low computation footprint which is consistent with our initial assumptions.

Fully Connected Layer 2

Table 25: FC2 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	437
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	33	-	0	0
Multiplexer	-	-	-	107
Register	-	-	105	-
Total	33	1	141	584
Available	1824	2520	548160	274080
Utilization (%)	1	~0	~0	~0

Layer	Inputs	Input Size	Outputs	Output Size	Latency (cycles)
Layer 2	1024	1024x1	1024	1024x1	198851

Fully connected layer FC2 has input dimension of 1 x 1024 and an output dimension of 1 x 1024. This implies that it stores 1024 x 1024 weight values for the matrix multiplication. In terms of performance, we see from table 25 that the design takes 198851 clock cycles to run. For resource utilization, we have a very small block RAM consumption of 33 blocks and a negligible utilization of LUTs and FFs. However, the RAM consumption is for the layer design and it does not take into consideration the network weights which are of the order 1024 x 1024.

Fully Connected Layer 3

Fully connected layer FC3 is the last layer and has input dimension of 1 x 1024 and an output dimension of 1 x 10. This implies that it stores 1024 x 10 weight values for the matrix multiplication. In terms of performance, we see from table 26 that the design takes 2500 clock cycles to run. For resource utilization, we have a very small block RAM consumption of 33 blocks and a negligible utilization of LUTs and FFs. However, the RAM consumption is for the layer design and it does not take into consideration the network weights which are of the order 10 x 1024.

Table 26: FC3 Requirements

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	428
FIFO	-	-	-	-
Instance	0	-	36	40
Memory	32	-	32	3
Multiplexer	-	-	-	113
Register	-	-	61	-
Total	32	1	129	584
Available	1824	2520	548160	274080
Utilization (%)	1	~0	~0	~0

Layer	**Inputs**	**Input Size**	**Outputs**	**Output Size**	**Latency (cycles)**
Layer 2	1024	1024x1	10	10x1	2500

Since the entire network is designed as a single pipeline, the maximum network performance is as good as the worst performing component of the network. For our design, convolution layer 2 had a latency of 4.6 million clock cycles. Since our design has block RAMs storing output from each layer, the design can be fully pipelined. Throughput is the number of classifications the network can perform in a second. To calculate the maximum possible throughput of the network, we have to use the latency of the worst performing layer which is 4.6 million clock cycles. We get the following final throughput for our design

Latency: 4.6 million cycles

Clock Frequency: 150 MHz

Images/sec: 33

Processing Time: 30 ms/image

CHAPTER 8

NETWORK BLOCK DESIGN

8.1 Introduction

We summarize the design using the Vivado block design tool. Independently created CNN layers or IP packages can be imported in the design tool to form a chain architecture which has been thoroughly explained in the previous section.

We explain the design components which are added along with the layer IP blocks to enable the FPGA functionality. Figure 43 shows an overview of block design.

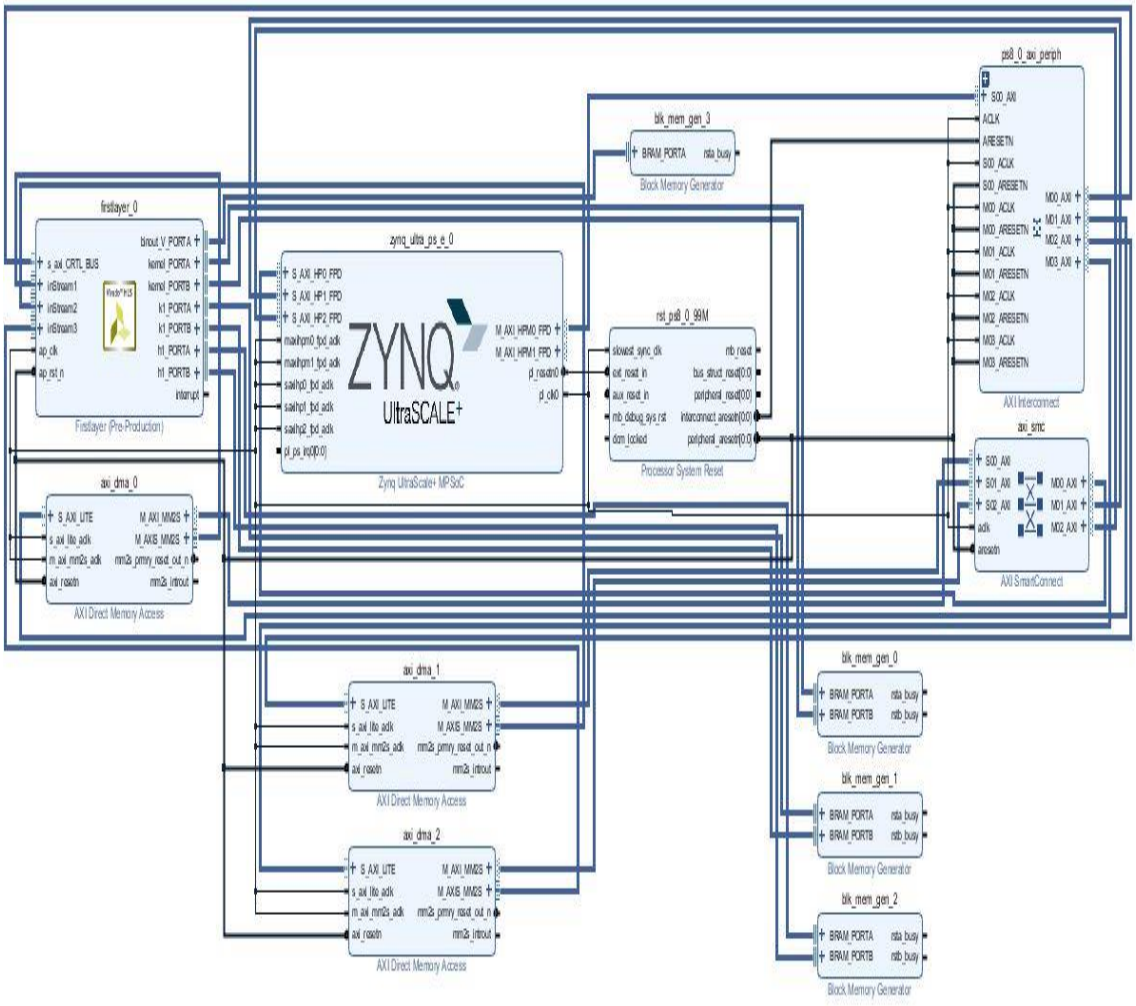


Figure 43: Overview of Block Design Complexity

8.2 Zynq Ultrascale

For sequence control, we can use the Zynq ultrascale controller provided on board. It has access to both programmable logic as well as DDR memory.

The Zynq block has a number of high performance (HP) ports which are used to transfer data from DDR to the PL via Axi DMA controller.

8.3 IP Block Ports

The custom designed layers of the network are imported individually as IP packages. Using appropriate pragmas in the HLS code, we declare the nature of the ports of the IP as per our requirements. As an example, we look at the IP block of the first layer (figure 44). The IP block contains a Control BUS (CTRL_BUS) which is used by the Zynq core to start stop and monitor the layer operation. This bus enables the sequence control of the network.

We can also see three input stream ports in the IP block which are used for high speed data transfer from off chip DDR to the programmable logic. These ports are used to acquire the three layers of the input feature map which in this case is the input image.

We have two port block RAMs available on chip which we will use in our design. For the output feature map, we declare it as a single port access since we need the second port of the BRAM to be used by the subsequent layer IP in the network (output from one layer is input to the next). For filter weights and normalization parameters, we can use true dual port configuration to enable faster operation of each layer. We demonstrate a single IP block design in figure 44.

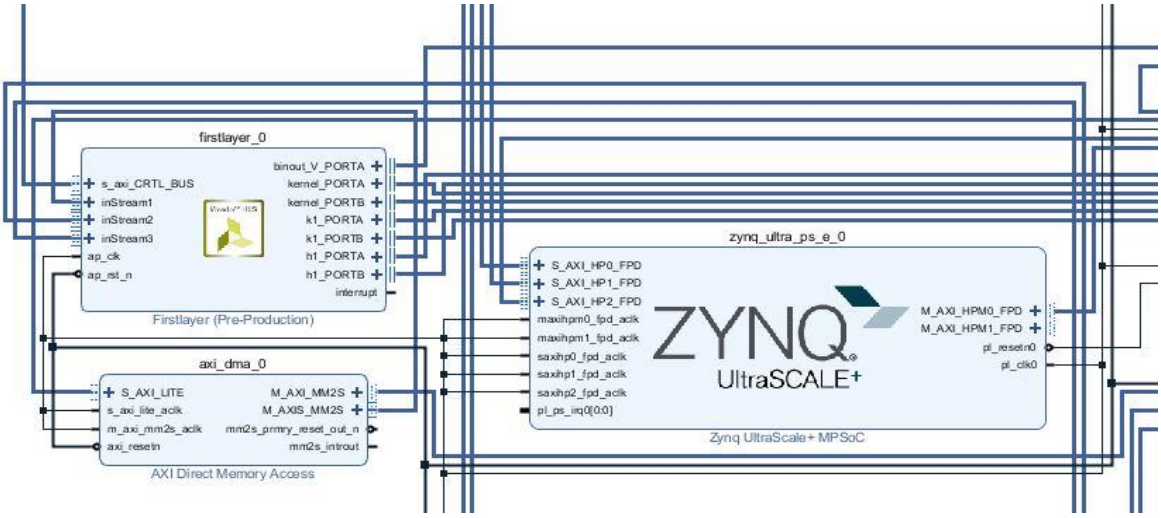


Figure 44: Expanded View for IP core Integration with ZYNQ Core in Block Design

8.4 Auxiliary Components

Direct Memory Access (DMA), AXI Interconnect and block memory generator are some of the essential auxiliary components of any design. These can be seen in figure below. Interconnect blocks form a gateway between our blocks. They imitate a slave port for a master port of an IP block and similarly present a master for a slave. They are used to connect multiple ports together. DMA provide the Zynq core with the capability to transfer data from off chip to programmable logic. HP ports of the core provide memory address of the DDR to be read and transferred as stream to the custom IP. Block memory generator block generates either single port of dual port block RAM in the design. It is also used for other important configurations such as standalone mode for sequential operation of layers without need for BRAM controller.

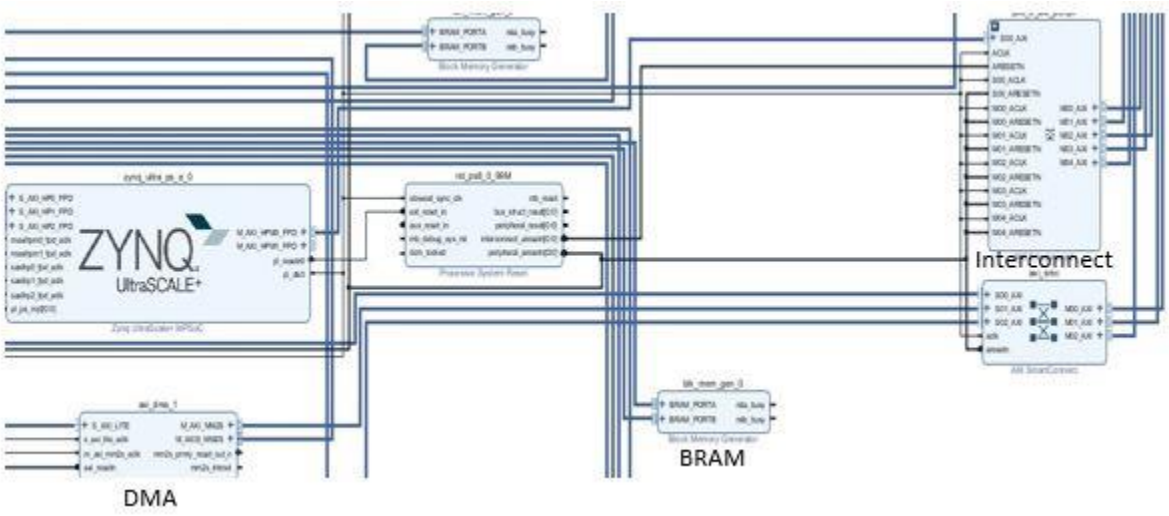


Figure 45: Expanded View for Auxiliary Components of Design

Figure 45 shows an overview of auxiliary components in block design. All the layer IP blocks are similarly added with their required auxiliary components to form the complete network which is then controlled through the Zynq core. This is the final stage of the development cycle. Extra functionality can be added such as timers for performance measurement and display port for output display on monitor for better visualization.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

In this work we have tried to achieve an optimal design for our deep network both from a software design perspective as well as the hardware implementation. We have seen an overview of the current trends in research on convolutional neural networks, components of these networks, variations in network designs and their performance.

We analyzed different layers of a deep network and established a relationship with the classification performance for the CIFAR-10 dataset. The design space for a deep convolutional network is infinitely large and requires in depth study to understand the correlation between parameter settings and performance. Different combinations of network layers and configuration were trained to calculate performance accuracy. Analysis of trained network performance showed the impact of convolutional layers and fully connected layers in increased performance of the network. We also deduced the vulnerability of network performance to extremely dense input layer of the fully connected side of the network. Filter size proved to be less impactful on the network performance albeit in a positive manner.

Final network implementation was carried out on hardware using Vivado HLS tool. Various design methodologies as well as coding schemes were tried. The vulnerability of design performance to coding style at a high level language such as C was very obvious from the results. Efficient resource allocation to a highly pipelined design proved to be the best solution to the performance issue of the network.

9.2 Future Work

This work provides a solid foundation of subsequent architecture development. Further exploration of neural network architecture design can be carried out which includes larger data sets such as ImageNet which has a lot more practical utility as compared to the smaller CIFAR-10 dataset. This also includes experimenting with deeper networks while at the same time keeping memory consumption at a minimum. There is also room in the betterment of layer design and architecture. The network latency should be lowered to increase performance especially if larger dataset classification is to be considered. Coding guidelines provided in this work can also help to achieve those targets. Ternary weight networks are another area which merit focus. Network proposed in this work can be extended to include ternary network with appropriate modifications.

REFERENCES

[1] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, *et al.*, "Accelerating binarized convolutional neural networks with software-programmable fpgas," *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp. 15-24.

[2] Google Brain Team, "TensorFlow," ed: Google, 2015.

[3] Y. Jia, "Caffe, Deep Learning Framework," ed: Berkeley Artificial Intelligence Research.

[4] A. Gray, "DIGITS: Deep Learning GPU Training System," in *NVIDIA Developer Blog* vol. 2018, ed, 2015.

[5] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *In Advances in neural information processing systems* pp. 3123-3131.

[6] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830,* 2016.

[7] X. Zynq, "7000," *Zynq-7000 all programmable soc overview, advance product specification-ds190 (v1. 2) available on: http://www. xilinx. com/support/documentation-/data sheets/-ds190-Zynq-7000-Overview. pdf," August,* 2012.

[8] V. Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis (UG902 (v2012.4))," PDF File2012.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," *In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp. 161-170.

[10] R. J. Schalkoff, *Artificial neural networks* vol. 1: McGraw-Hill New York, 1997.

[11] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," *In Field Programmable Logic and Applications, 2009. FPL 2009* pp. 32-37.

[12] C. Rosenberg. (2013, 14 May). *Improving Photo Search: A Step Across the Semantic Gap*. Available: https://ai.googleblog.com/2013/06/improving-photo-search-step-across.html

[13] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, pp. 221-231, 2013.

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[15] H. Yonekawa and H. Nakahara, "On-Chip Memory Based Binarized Convolutional Deep Neural Network Applying Batch Normalization Free Technique on an FPGA," *In Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International* pp. 98-105.

[16] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, pp. 141-142, 2012.

[17] Y. LeCun, "LeNet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, p. 20, 2015.

[18] A. Krizhevsky. (2017, 14 May). *The CIFAR-10 dataset*. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," *In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* pp. 248-255.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* pp. 1097-1105.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, *et al.*, "Going deeper with convolutions." *Cvpr, 2015*.

[23] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp. 26-35.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[26] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[27] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp. 16-25.

[28] J. Kraus. (2013, 13 May). *An Introduction to CUDA-Aware MPI*. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[29] NVIDIA, "GEFORCE GTX 1080 User Guide," ed: NVIDIA Corporation, 2016.

[30] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[31] Xilinx, "Zynq-7000 All Programmable SoC Data Sheet: Overview," ed: Xilinx, 2017.

[32] NVIDIA. (2018, 3rd June). *NVIDIA TITAN X*. Available: https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/

[33] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded FPGA," *In Field Programmable Logic and Applications (FPL), 2017 27th International Conference* pp. 1-4.

[34] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* pp. 45-54.

[35] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072-1086, 2018.

[36] J. Wang, J. Lin, and Z. Wang, "Efficient Hardware Architectures for Deep Convolutional Neural Network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2017.

[37] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," *In Field Programmable Logic and Applications (FPL), 2016 26th International Conference on* pp. 1-8.

[38] Xilinx, "Zynq UltraScale+ MPSoC Data Sheet: Overview," ed: Xilinx, 2017.

[39] Xilinx, "UltraFast High-Level Productivity Design Methodology Guide UG1197," 2018.

[40] SDSoC Environment. (2017, 13 May). *pragma HLS interface*. Available: https://china.xilinx.com/html_docs/xilinx2017_1/sdsoc_doc/topics/pragmas/ref-pragma_HLS_interface.html

BIOGRAPHICAL SKETCH

Muhammad Mohid Nabil was born in Rawalpindi, Pakistan in November 1990. He completed his high school among top 25 students of the state. He went on to obtain his Bachelor of Science in Electrical Engineering from College of Electrical and Mechanical Engineering, NUST in 2013. He started his Master of Science from The University of Texas at Dallas in January of 2016. During the course of his Master's degree, he did research work in the Cochlear Implant Laboratory as well as IDEA Lab. He is passionate about research on FPGAs, Deep Learning and Signal Processing.

CURRICULUM VITAE

Muhammad Mohid Nabil

EDUCATION

University of Texas at Dallas, USA Jan 2016- June 2018
Master of Science Electrical Engineering

National University of Science and Technology, Pakistan Sep 2009 – June 2013
Bachelor of Science Electrical Engineering

WORK EXPERIENCE

Instrumentation Engineer

ENGRO Corporation June 2013-Dec 2015

- Maintenance and troubleshooting for plant control systems and instrumentation
- Project proposals and execution for continuous process improvement
- Expertise with a variety of control systems including Siemens, Honeywell, ABB and Rockwell

RESEARCH EXPERIENCE

Master's Thesis Student

Integrated Design Engineering and Algorithms (IDEA) Lab Jul 2017 – June 2018

- Software and hardware implementation of low precision Binarized Neural Networks on Zynq architecture (ZCU102)
- Hardware components including 2D Convolution, Normalization and Binarization being designed using Xilinx Vivado HLS on FPGA coprocessor for high degree of parallelism
- Sequence control and data movement routines using embedded C on ARM core for optimal performance

Graduate Student Worker

Cochlear Implant Laboratory, University of Texas at Dallas Nov 2016-Jul 2017

- Using discrete time signal processing, physiological imaging and acoustics for perception measurements
- EEG signal acquisition and processing to establish neural correlate of audio stimuli
- FIR/IIR digital filter design using C++ to model speech enhancement in cochlear implant embedded DSP
- Performed human subject testing for audio quality evaluations using novel test schemes on Matlab/C++