ASIC Based Neural Network Accelerators

Final Year Project Report

Presented

by

Amna Arshad Khan
CIIT/FA19-EEE-037/ISB

Mohsin Iftikhar
CIIT/ FA19-EEE-036/ISB

In Partial Fulfillment

of the Requirement for the Degree of

Bachelor of Science in Electrical (Electronics) Engineering

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COMSATS UNIVERSITY ISLAMABAD

July 2023

ASIC Based Neural Network Accelerators

Final Year Project Report

Presented

by

Amna Arshad Khan
CIIT/FA19-EEE-037/ISB

Mohsin Iftikhar
CIIT/ FA19-EEE-036/ISB

In Partial Fulfillment

of the Requirement for the Degree of

Bachelor of Science in Electrical (Electronics) Engineering

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COMSATS UNIVERSITY ISLAMABAD
July 2023

Declaration

We, hereby declare that this project neither as a whole nor as a part there of has been copied out from any source. It is further declared that we have developed this project and the accompanied report entirely on the basis of our personal efforts made under the sincere guidance of our supervisor. No portion of the work presented in this report has been submitted in the support of any other degree or qualification of this or any other University or Institute of learning, if found we shall stand responsible.

Signature:_______________
Name Amna Arshad Khan

Signature:_______________
Name Mohsin Iftikhar

COMSATS UNIVERSITY ISLAMABAD
July 2023

ASIC Based Neural Network Accelerators

An Undergraduate Final Year Project Report submitted to the Department of

ELECTRICAL AND COMPUTER ENGINEERING

As a Partial Fulfillment for the award of Degree

Bachelor of Science in Electrical (Electronics) Engineering

by

Name	**Registration Number**
Amna Arshad Khan	CIIT/FA19-EEE-037/ISB
Mohsin Iftikhar	CIIT/FA19-EEE-036/ISB

Supervised by

Dr. Sikandar Gul
Assistant Professor
Department of Electrical and Computer Engineering
CU Islamabad

Dr. Rizwan Azam
Assistant Professor,
Department of Electrical and Computer Engineering
CU Islamabad

COMSATS UNIVERSITY ISLAMABAD
July 2023

Final Approval

This Project Titled

ASIC Based Neural Network Accelerators

Submitted for the Degree of

Bachelor of Science in Electrical(Electronics) Engineering

by

Name	**Registration Number**
Amna Arshad Khan	CIIT/FA19-EEE-037/ISB
Mohsin Iftikhar	CIIT/FA19-EEE-036/ISB

has been approved for

COMSATS UNIVERSITY ISLAMABAD

Supervisor -1	*Co-Supervisor-2*
Dr.Sikandar Gul	*Dr. Rizwan Azam*
Assistant Professor	*Assistant Professor*

Internal Examiner-1	*Internal Examiner-2*
Asma Jadoon	*Dr. Fawad Zaman*
Lecturer	*Associate Professor*

External Examiner
Dr Shabbir Majeed
Professor

Head
Department of Electrical and Computer Engineering

Dedication

We dedicate our final year project to our beloved family and friends for their prayers, care, and moral support.

Acknowledgements

All the prayers and trillions of humble thanks to Allah Almighty, for He blessed us with the courage to work on our thesis. Peace and blessings be upon Muhammad (S.A.W.). Indeed, Allah has been very kind to us throughout our lives. He blessed us with loving and supportive parents, great and competent teachers, and caring friends.

We wish to express our humble gratitude to our parents and siblings for their love, care, prayers, and moral support. Special thanks to our supervisors, Dr. Saad Arslan, Dr. Sikandar Gul (Assistant Professor, COMSATS University Islamabad), and Dr. Rizwan Azam (Assistant Professor, COMSATS University Islamabad), who have been kind to extend their valuable guidance and suggestions from the very start of the thesis. They deserve the credit; may Allah the Almighty bless them with countless favors both here and in the hereafter.

Amna Arshad Khan
Mohsin Iftikhar

Table of Contents

List of Figures

Abstract

Deep neural networks are receiving a lot of attention from the research community due to their extensive implementation. DNN model complexity often grows with application complexity, and the deployment of complicated DNN models requires a large amount of processing capacity. Complex DNNs cannot be processed by general-purpose processors within the required throughput, latency, and power consumption. As a result, domain-specific hardware accelerators must deliver significant computational resources while sustaining performance efficiency and throughput on a tiny chip area. The Convolutional Neural Network (CNN) is one of the most widely used neural network (NN) models as an efficient algorithm used in image processing, pattern recognition, and a variety of other real-world applications. However, they can be computationally expensive to train and deploy. In this thesis, we propose a new method for designing CNN hardware accelerators that can achieve high performance and energy efficiency. Our approach is based on a deep understanding of the CNN architecture. We first train a CNN model on the MNIST dataset using Tensor Flow, then implemented this same model in native Python without the use of any external libraries. This has helped us to understand the inner workings of the CNN model in detail. We are using this knowledge to design a new hardware accelerator architecture. Our aim for the hardware accelerator is to achieve the same accuracy as Tensor Flow and native Python code have achieved on the MNIST dataset, while consuming less power. The method that we have planned for this project is, firstly, we will train our model after receiving satisfactory performance, it can be deployed for real-world inference tasks. We believe that this approach can be used to design efficient CNN hardware accelerators for a wide range of applications.

1. Introduction

1.1 Purpose

ASIC-based neural network accelerators are custom-designed chips that offer improved performance, reduced power consumption, increased scalability, and lower costs compared to general-purpose processors. They excel in deep learning tasks, such as image classification, natural language processing, and speech recognition, by efficiently executing the specific operations required for these applications. With faster processing, lower power requirements, and the ability to scale for complex models, ASIC-based accelerators are an ideal solution for a wide range of deep learning applications. Additionally, their lower manufacturing costs make them an attractive option for implementing deep neural networks at scale.

1.2 Background

Neural networks are a group of algorithms that work like the human brain. When you look at something, your brain uses neurons to process the information and recognize what the object it is. Neural networks do something similar. They take in a lot of data, find patterns in it, and then give an output.

The brain processes a lot of information when it sees an object. Each neuron in the brain is connected to others and works in its own area of the brain. This is like how each neuron in a neural network works in its own area of the data. The layers in the neural network detect simpler patterns first, like lines and curves, and then more complex patterns, like faces and objects. This is how computers can "see" things by using neural networks.

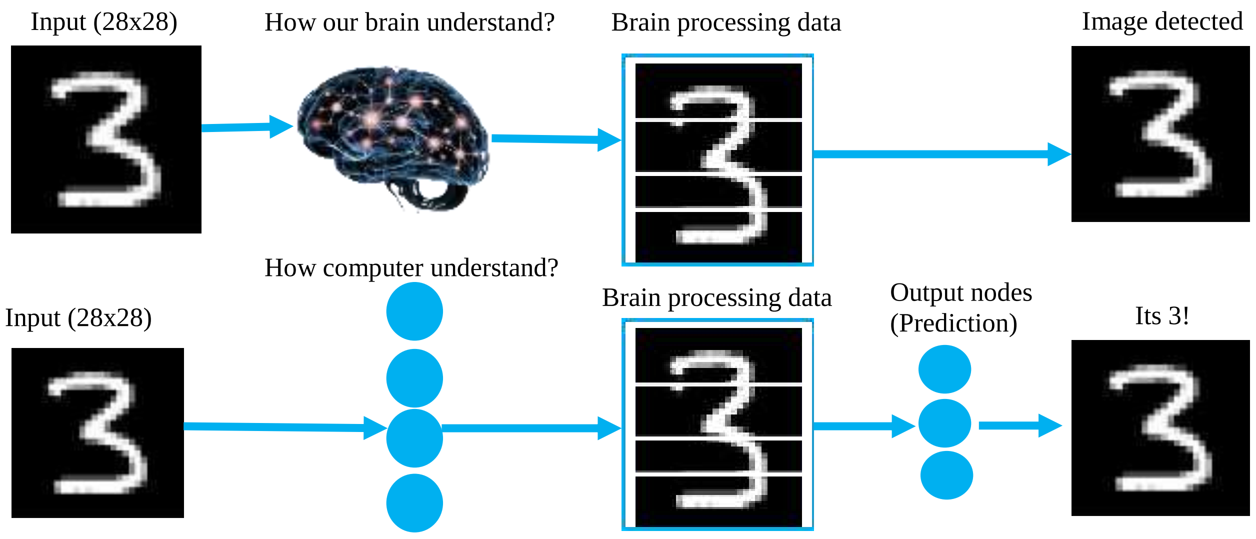


Figure 1.1: Similarity of the brain with ANN in processing an image

The figure 1 elaborates on the working of neurons in the brain, links their working with that Artificial Neural Networks, and shows how they are similar in their working.

Neural networks, also known as artificial neural networks (ANNs), are computer models that imitate the way the human brain functions. These networks consist of many connected processing elements, or neurons, that work together to solve specific problems. One type of neural network is a deep learning network, which uses layers of algorithms to create an artificial neural network (ANN) that can learn and make accurate decisions on its own.

ANNs are used to process information and learn from data, just like humans learn from experience. They are used for various applications like image recognition [7], voice recognition, and robotics, among others. ANNs have a specific architecture format, inspired by the biological nervous system, with neurons that are connected to each other by weighted links. The network learns through a process of training that involves computations and mathematics that simulate the human brain. Convolutional neural networks are one type of ANN that is particularly used for image processing.

1.3 Introduction of Convolutional Neural Networks (CNN)

CNNs use convolutional layers to analyze the input image, breaking it down into smaller, simpler features, like lines and edges. These features are then combined in subsequent layers to form more complex features, like shapes and textures until the network can recognize the entire object in the image.

Traditional neural networks can also recognize images, but they become slow and require a lot of power when dealing with larger images due to the number of parameters involved. CNNs specialize in processing image data, which is represented by pixels arranged in a grid-like fashion.

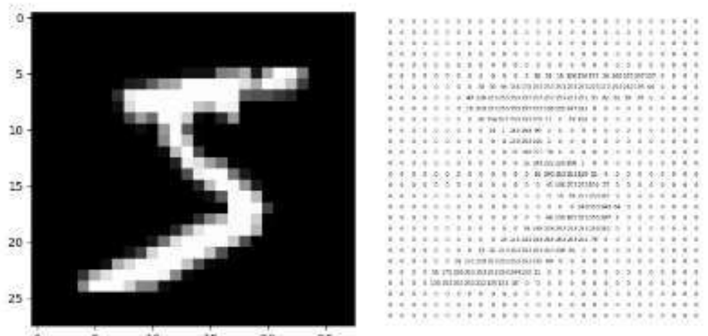


Figure 1.2: Image representation as a grid of pixels

1.4 Objectives

- Design and evaluation methodology for the accelerator architecture.
- Comparing the computation results of the hardware against the model.
- Obtaining efficient computational cost (low-power consumption, less computational time).
- Chip implementation by converting a high-level CNN to a hardware structure.

1.5 Project Scope

The design of ASIC based neural network accelerator can efficiently support a wide variety of neural network architectures, minimize the power consumption of the accelerator and maximizing the performance of the accelerator.

1.6 Problem Statement

Deep neural networks (DNNs) are computationally expensive to train and deploy. ASIC-based neural network accelerators can address this challenge by providing significant improvements in power consumption and performance. Our project aims to develop an ASIC-based neural network accelerator for both training and inference that is low-power and compact, making it ideal for a wide range of computing applications.

1.7 Broader Impact (UN SDGs)

There are a number of factors to take into account when assessing how an ASIC-based neural network accelerator project may affect the Sustainable Development Goals (SDGs) of the United Nations. Here are a few possible areas of influence, each with an explanation.

1.7.1 Targeted SDGs:

The targeted sustainable development goals of our project for United Nation are:

SDG 4: Good Health and Well Being:

The project aim is to ensure healthy lives and well-being for all the ages. ASIC based accelerators can be deploy in medical applications that can help to diagnose diseases, monitor patients and deliver the personalized treatments. For example, ASIC-based neural network accelerators can be used to develop systems that can automatically detect cancer cells in medical images, can be fit into MRI and CT-Scan machines which can reduce the time significantly or systems that can monitor patients vital signs in real time.

SDG 4: Quality Education:

The project advances technologies in the areas of artificial intelligence (AI) and deep learning by creating ASIC-based neural network accelerators. By making it possible for more effective and potent AI-driven instructional tools, these developments can raise the quality of education. All learners will benefit from using these tools since they can offer personalized learning experiences, flexible tutoring programs, and intelligent content recommendation algorithms.

SDG 8: Decent work and Economic growth:

The deployment of ASIC in industrial applications can create a new jobs and boost economy growth with full and productive employment and decent work for all. For example ASIC-based neural network accelerators can be used to develop robotics and automation systems that can automate tasks that are currently performed by humans, or systems that can improve the efficiency of manufacturing processes.

SDG 9: Industry, Innovation, and Infrastructure:

The effectiveness and performance of AI applications can be greatly enhanced using ASIC-based neural network accelerators. These accelerators make neural network processing faster and more energy-efficient, that can improve the quality of life and create a more sustainable future. For example, ASIC-based neural network accelerators can be used to develop smart cities systems that can manage traffic, energy, and other resources more efficiently, or systems that can monitor and protect the environment.

2. Literature Review

Researchers from both academia and industry are actively working on constructing high performance hardware for Deep Neural Network (DNN) inference [1]. The integration of DNNs in real-time applications necessitates low-power and high-throughput DNN accelerators. In recent years, several high performing DNN accelerators architecture have been suggested to enhance overall DNN performance. Customized domain-specific accelerators can further improve performance by tailoring the architecture to specific applications. Graphics Processing Units (GPUs) offer massive parallel computing capabilities and excel at parallel DNN computations. However, their high-power consumption limits their usage in embedded systems. Field-Programmable Gate Arrays (FPGAs) provide high performance and are often employed for prototyping and design verification. Application-Specific Integrated Circuits (ASICs) are custom-designed for specific applications, offering optimal speed and power consumption. ASICs find multiple applications in embedded devices but have long development cycles and lack flexibility after design completion. The focus has shifted towards ASIC accelerators due to limited control and flexibility in designing Multiply-Accumulate (MAC) units in FPGAs. Nevertheless, recent advancements have explored precision-scalable MAC units in different accelerators and highlighted their advantages in various scenarios. While MAC units play a crucial role in improving accelerator performance, other factors, such as data flow and on-chip memory, also impact overall efficiency. An efficient architecture should strive for maximum MAC utilization, aiming for 100% utilization. Existing research typically employs metrics like chip area, throughput, latency, and energy efficiency for performance comparisons. However, an accelerator specifically designed for a particular DNN model, taking into consideration factors like sparsity and kernel size, may not uniformly offer comparable benefits for other DNN models.

The primary objective of the proposed accelerator is to design a low-area and power-efficient hardware accelerator with optimal performance [2]. While there are existing hardware accelerators, our focus is on creating a highly efficient and accurate accelerator. With the increasing complexity of neural network models, the area and power requirements also tend to grow. Therefore, it becomes crucial to employ techniques that can reduce area, power, and energy consumption while maintaining efficiency. To address this, we developed an optimized Convolutional Neural Network (CNN) architecture that is shallower, utilizes fewer filters, and incorporates narrower bit widths for weight parameters, all while ensuring high accuracy.

To demonstrate the effectiveness of our proposed architecture and design technique, we conducted experiments using a small CNN and the MNIST dataset for training and validation. Our target was to achieve an accuracy of 95% or higher. Throughout our research, we have recognized the significance of understanding the factors that influence performance in DNN accelerators. It is essential to develop energy and performance-efficient accelerators to meet the demands of various applications.

Building upon our current work [2], our future plans involve developing a flexible architecture that can be utilized to accelerate different applications within the network. This flexibility will enable us to enhance the versatility and adaptability of the accelerator for a wide range of neural network tasks.

A reconfigurable, FPGA-based CNN accelerator for object recognition applications was proposed by Li et al. [3]. It balances parallel computing capability and system power consumption by using a kernel partition strategy to minimize recurrent usage to input feature maps and kernels. At 151.4 frames per second for the AlexNet, the proposed CNN accelerator reaches an optimum throughput of 220.0 GOP/s with a 22.9 GOPs/W energy efficiency. Additionally, it may be changed to process VGG-16 for the complex recognition of objects.

Convolutional neural networks (CNNs) are being accelerated using FPGA technology, and Ma et al. [4] have explored this progress and provided a performance model to predict the performance and resource use of an FPGA implementation. A number of CNN algorithms are used to validate the suggested model, and the outcomes are compared to the onboard test results on the two separate FPGAs. The outcomes demonstrate that the suggested performance model is capable of properly predicting the performance and resource use of the FPGA implementation as well as early design bounds and performance bottleneck identification. The literature review section [4] covers pertinent research that has optimized the computation patterns and memory access of their suggested architecture and data flow by using performance models. Convolution was implemented by Suda et al. [5] as matrix multiplication, and the design was improved using a performance model. The execution time in [5] does not take the DRAM transfer delay into account; it solely counts calculation time. The model in [5] is unable to accurately estimate the total latency as a result of the design being memory-bounded, leading to the estimation mismatch of fully linked layers with high compute parallelism. Using a performance model, the suggested systolic array design in [6] is also made more efficient.

2.1 Literature Review Table

Paper	Year	Topic	Methodology	Results	Future Work
"FPGA-Based High Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio"	2022	FPGA-based CNN accelerator	Novel architecture for both convolutional and fully connected layers	Achieved high throughput and resource utilization	Further optimization of the architecture, exploration of new CNN architectures, and application to other machine learning tasks
"An FPGA-Based Energy-Efficient Reconfigurable Convolutional Neural Network Accelerator for Object Recognition Applications"	2022	FPGA-based CNN accelerator	Kernel partition technique	Achieved high throughput and energy efficiency	Further optimization of the architecture, exploration of new CNN architectures, and application to other machine learning tasks
"Design of Power-Efficient Training Accelerator for Convolution Neural Networks"	2021	FPGA-based CNN training accelerator	Resource sharing, integrated convolution-pooling block, concurrent floating-point data paths	Achieved high throughput and energy efficiency	"Design of Power-Efficient Training Accelerator for Convolution Neural Networks"
"Implementation of Convolutional Neural Networks in FPGA for Image Classification"	2019	FPGA-based CNN accelerator	Scalable and modular architecture	Achieved high throughput and energy efficiency	"Implementation of Convolutional Neural Networks in FPGA for Image Classification"

"FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review"	2018	FPGA-based CNN accelerator	Review of existing techniques	Highlights key features and provides recommendations	
"Recent Developments in Low Power Machine Learning Hardware Accelerators for Mobile Devices"	2018	Low power machine learning hardware accelerators for mobile devices	Dataflow, reduced precision, model compression, and sparsity	Achieved high efficiency	Explore new techniques for reducing energy costs and improving performance
"An FPGA 2D Convolution unit based on the CAPH language"	2017	FPGA-based 2D convolution	CAPH language, parallel convolution	Achieved high performance, reduced hardware resource consumption	
"Design and Implementation of Hardware Accelerator for Deep Convolutional Neural Networks"	2018	FPGA-based CNN accelerator	Dataflow optimization, internal parallelism reduction	Achieved high efficiency	Increase on-chip memory size, try bigger filter kernels, test in a particular framework

3. Methodology

This chapter provides a comprehensive overview of the methodology employed in our project. It includes essential definitions of key terms and concepts that will be frequently referenced throughout the study.

3.1 Block Diagram of Accelerator

The following block diagram shows the high-level architecture of a Python native model for an ASIC-based neural network accelerator which is further translated to the hardware description language HDL. The HDL language will be synthesized and will be implemented on a chip

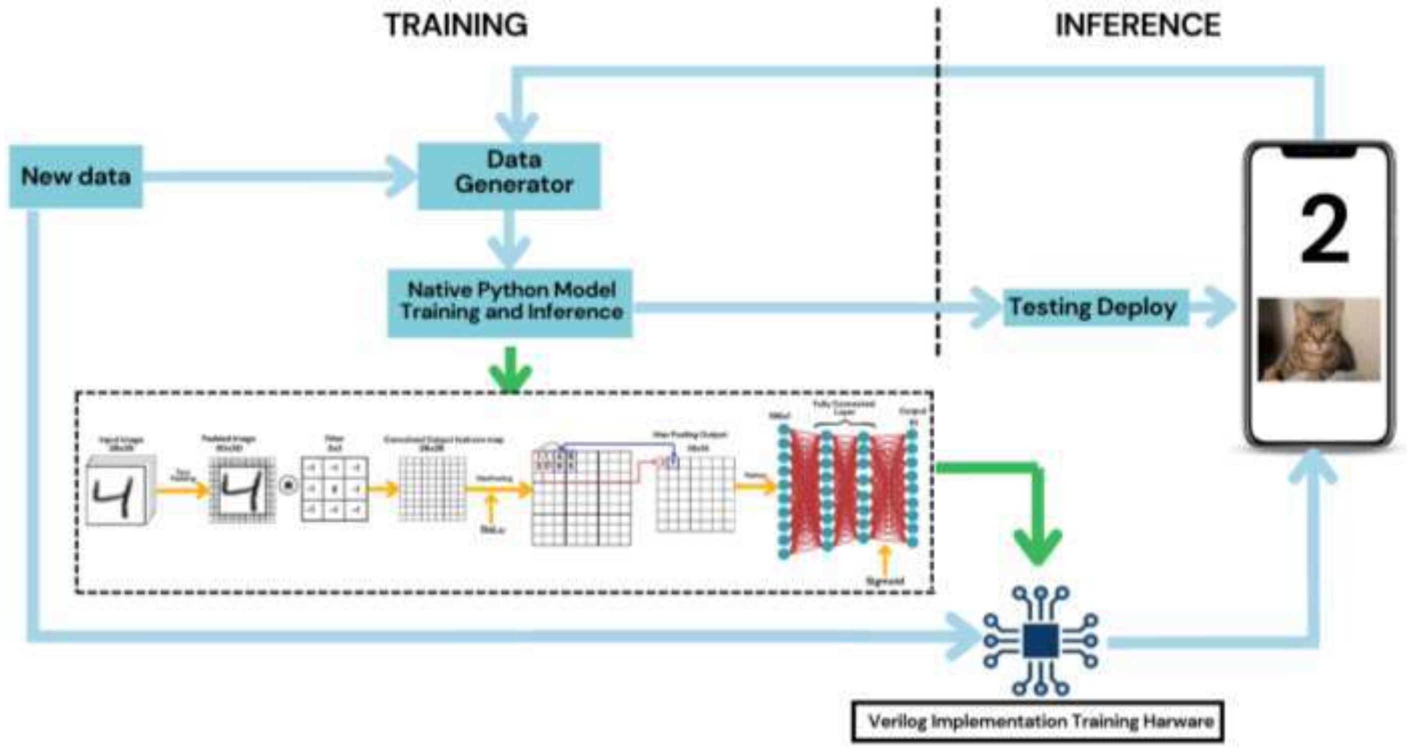

TRAINING
INFERENCE
New data
Data Generator
Native Python Model Training and Inference
Testing Deploy
2
Verilog Implementation Training Harware

Figure 3.1: Block Diagram of Proposed Design

Types of Layers in a convolutional neural network

1. Convolution Layer.
2. Pooling
3. Fully Connected

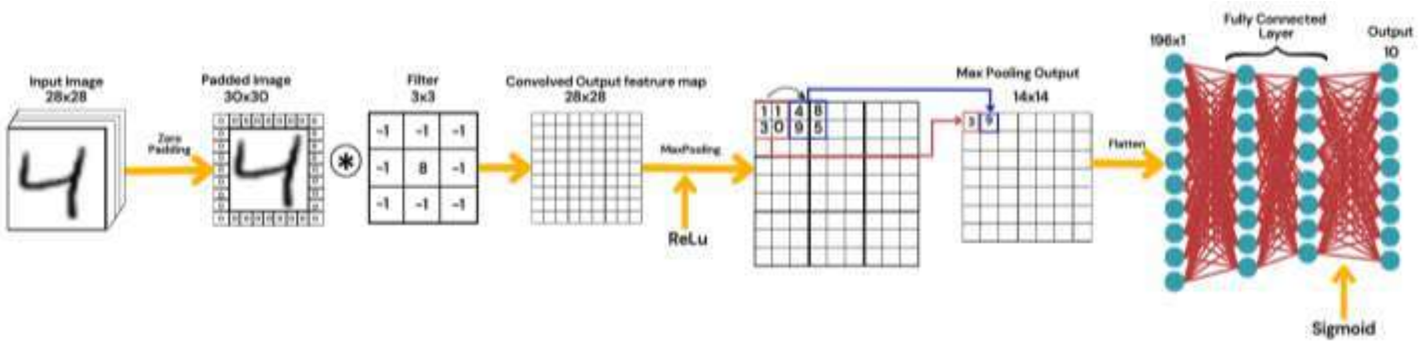


Figure3.2: Convolution Neural Network Architecture

An architecture that conducts both the training and inference in a single chip can provide significant performance and energy sufficient benefits as compared to the traditional architecture that require separate hardware for different tasks.

3.2 Method for Neural Networks

For making our model train training and inference plays a very important role.

3.2.1 Training:

It refers to the process of updating the weights and biases of neural network using a set of labelled data to minimize the error between the predicted outputs and true label. Training a neural network is typically a compute-intensive process that involves many iterations of forward and backward propagation.

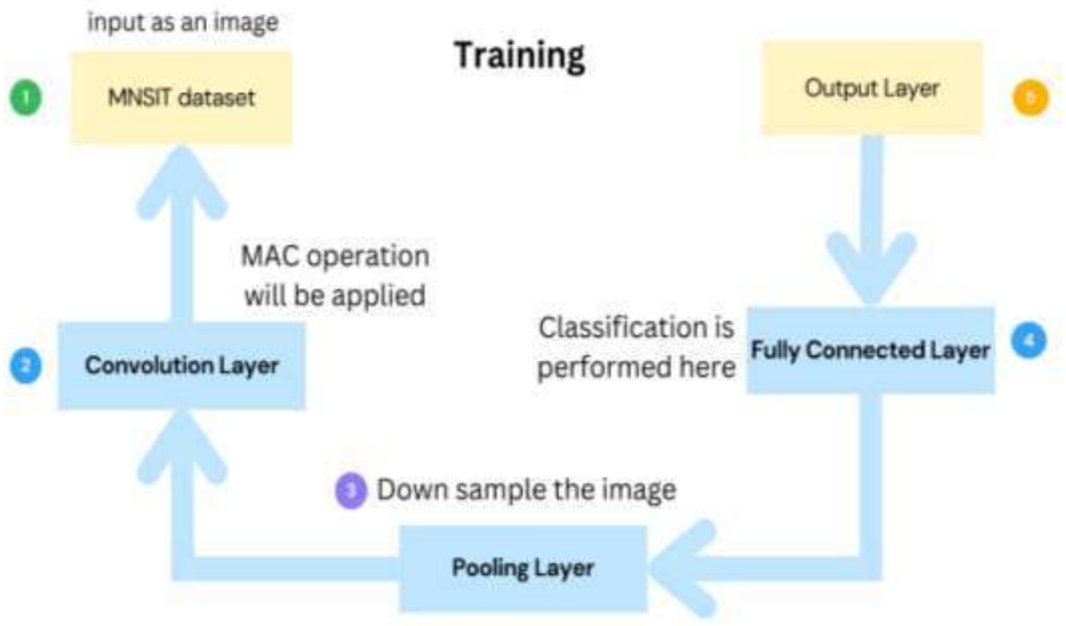


Figure 3.3: Logical Diagram of Training a Neural Networks

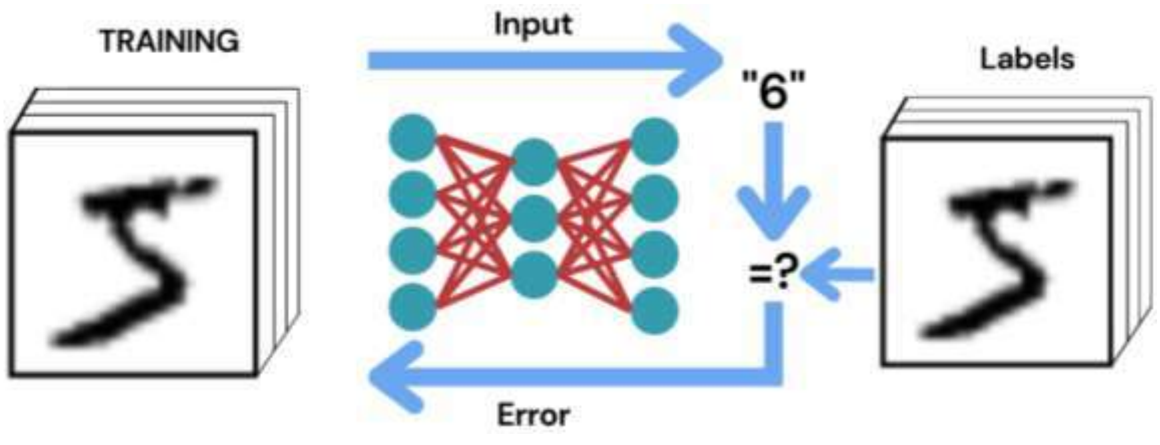


Figure 3.4: Working Block Diagram of Training in Neural Networks

3.2.2 Inference:

Inference refers to the process using a trained neural network to make predictions or decision based on input data. During inference, the input data is fed into the neural network and the neural network and then network computes the output using the weights and biases that a model learned during training.

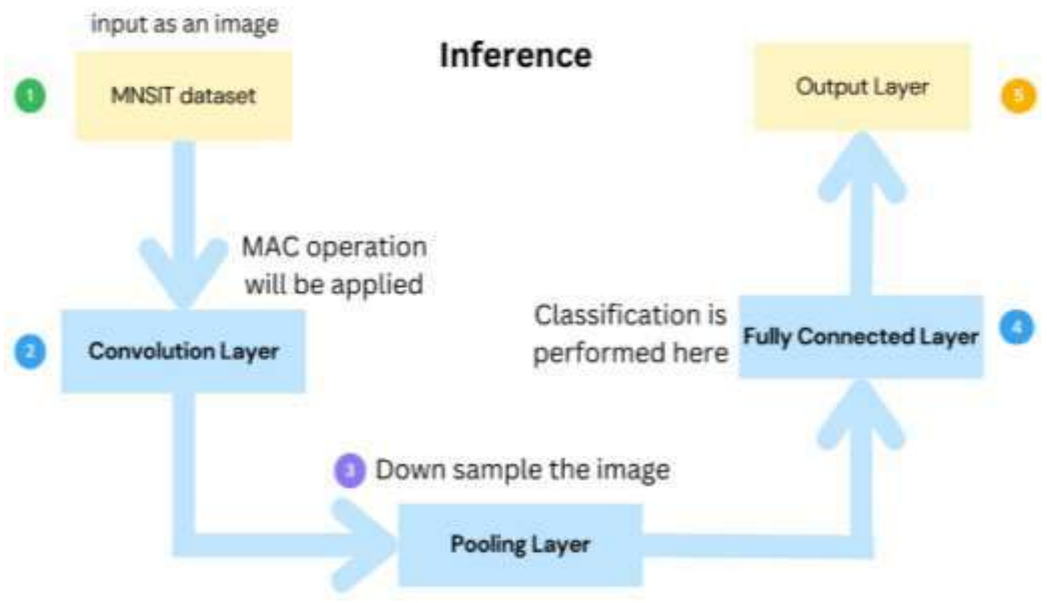


Figure 3.5: Logical Diagram of Inference in Neural Network

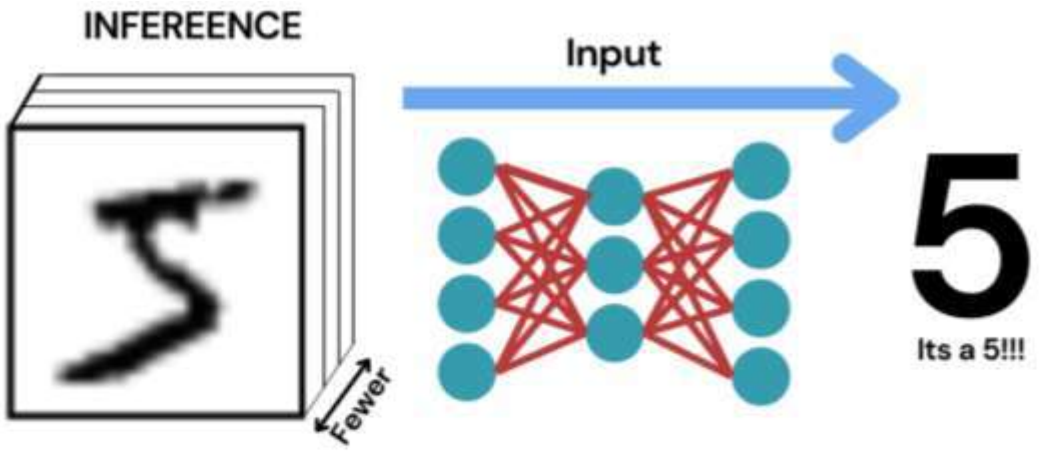


Figure 3.6: Working Block Diagram of Inference in Neural Networks

Here is an analogy that can clear the difference between training and inference. Imagine that you are training a student to become a doctor. The training process would involve teaching the student about anatomy, physiology, and pharmacology. The student would also need to practice performing surgery on cadavers. This training process would be long and difficult, but it would be necessary for the student to become a competent doctor.

Once the student has graduated from medical school, they can begin practicing medicine. This is the inference phase. When a patient comes to see the doctor, the doctor will use their knowledge and skills to diagnose the patient's illness and recommend a course of treatment. This process is much less time-consuming than the training process, but it is still essential for the doctor to be able to provide quality care to their patients.

In the same way, training a neural network is a long and computationally intensive process, but it is necessary for the neural network to be able to perform inference tasks in real time.

3.3 Essential Methodology Steps for Success

This methodology is followed to develop ASIC-based neural network accelerators that can achieve high performance and energy efficiency.

1. Developing Python model with training and inference:

To develop a Python model with training and inference capabilities, we can use popular deep learning frameworks like TensorFlow, PyTorch, or Keras. However, for a more in-depth understanding of the calculations of every layer and to help with future hardware design, we can build our own Convolutional Neural Network (CNN) architecture using native Python code. This involves defining the structure of the neural network, including the number of layers, neurons, and activation functions. Once the architecture is defined, we can train the model using a large labeled dataset, which may take considerable time. After training, we evaluate the model's performance using a separate test dataset, measuring metrics such as accuracy and precision. Defining the architecture, training the model, and evaluating its performance are crucial steps in building a Python model with training and inference capabilities, allowing us to gain a deeper understanding of the neural network's inner workings and paving the way for potential hardware design advancements.

2. Analyze HW requirements such as bandwidth, memory and design partitioning:

When analyzing the hardware requirements for a neural network accelerator, there are a few important factors to consider. Firstly, the size and complexity of the neural network architecture have a big impact on the memory and bandwidth needed for the accelerator. If the architecture is more complex with lots of layers and neurons, it will require more memory and bandwidth. Secondly, the size of the dataset used for training and inference also affects the memory requirements. Bigger datasets need more memory capacity. Lastly, the desired performance level of the accelerator determines the necessary bandwidth. Higher performance goals call for more bandwidth. By taking these factors into account, we can design the hardware accelerator by optimizing performance and area through proper partitioning.

3. Translate python model to Verilog HDL, while being aware of the HW requirements:

We have written the Verilog HDL code that is manual coding, where you write the Verilog HDL code for each part of the neural network accelerator by hand. This gives you complete control over the design but takes more time. After translating the Python model to Verilog

HDL, you can simulate the model to check its accuracy and synthesize it to create a hardware implementation.

4. Implement HW accelerator design using Verilog HDL:

To implement a hardware accelerator design using Verilog HDL, you need to follow a few key steps. Firstly, you write the Verilog HDL code for the different blocks of the neural network accelerator, either manually or using an automatic code generation tool. This step allows you to define the behavior and structure of the accelerator. Next, you synthesize the Verilog HDL code, which involves transforming it into a gate-level netlist. This synthesis process optimizes the design for factors like area, speed, and power consumption. Finally, you perform place and route, where the gate-level netlist is transformed into a physical chip. This involves placing the components on the chip and establishing the necessary connections. Once the hardware accelerator design is implemented, you can test it using a simulator or deploy it on a physical chip.

5. Test and Verify HW accelerator:

To test and verify a hardware accelerator, there are several steps you can follow. First, you can simulate the design using a simulator, which is a cost-effective way to check if the accelerator functions correctly. The simulator runs the design on a virtual platform and provides a report to verify its accuracy. Additionally, you can fabricate the design and test it on a physical chip. This involves creating a chip from the design and conducting various tests, such as checking its functionality, performance, and power consumption. It's important to use a combination of testing methods, such as simulation, emulation (which is faster but more expensive), and physical testing (which is the most accurate but slower). Once the hardware accelerator is successfully tested and verified, it can be deployed in a production environment. To ensure thorough evaluation, use a reliable simulator, select a representative physical chip, test under different conditions, and collect data for analysis and troubleshooting.

4. Fundamentals of Convolutional Neural Networks

Before going into the details of the convolutional neural network layers, it’s necessary to cover some basic definitions of image, filter, stride, padding and convolution.

Image:

An image is represented in two or three-dimensional form. It is made up of pixels arranged in columns and rows, forming a matrix or an array. In a grayscale image, each pixel is an integer value between 0 (black) and 255 (white), giving 256 different shades of gray in an 8-bit color format. In the binary image, there are only two-pixel values, 0 and 1, representing black and white, respectively.

Filter:

The filter size is smaller than the input data size, typically 3x3. The filters slide across the input data, performing dot product calculations between the pixels. A dot product is an element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the “scalar product“.

We have multiple filters:

-1	-1	-1
-1	8	-1
-1	-1	-1

(1)

0	-1	0
-1	5	-1
0	-1	0

(2)

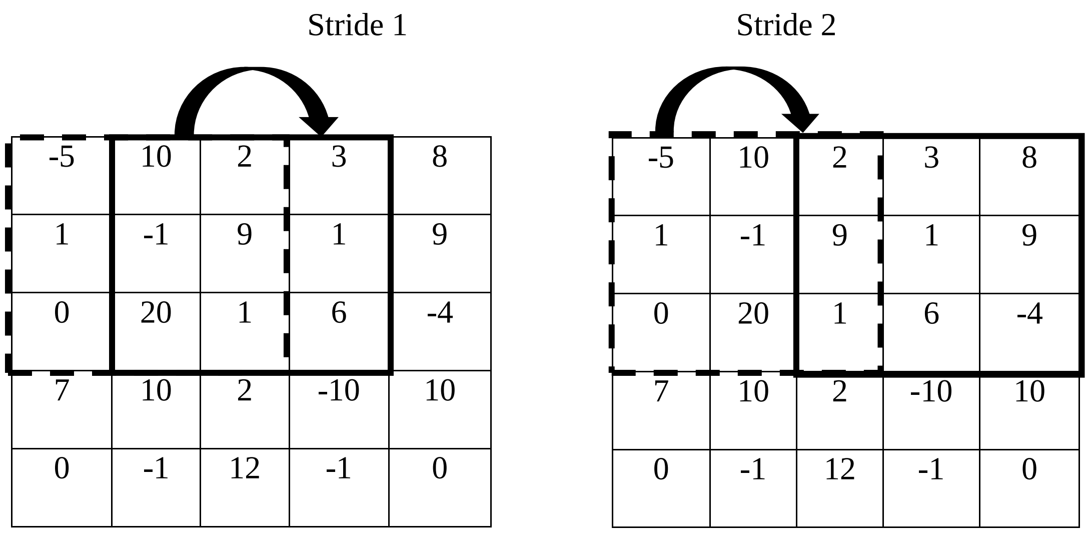
0	-1	0
-1	5	-1
0	-1	0

(3)

Figure 4.1: Filters (1) Edge Detection (2) Sharpening Filter (3) Laplacian Filter

Stride:

Stride refers to the number of pixels the filter moves at a time while scanning through the input image during the convolution operation. It determines the distance between the placements of the filters on the image. If the stride value is large, then the output feature map will be smaller, and if the stride value is small, the output feature map will be larger.



-5	10	2	3	8
1	-1	9	1	9
0	20	1	6	-4
7	10	2	-10	10
0	-1	12	-1	0

-5	10	2	3	8
1	-1	9	1	9
0	20	1	6	-4
7	10	2	-10	10
0	-1	12	-1	0

Figure 4.2: Process of the Kernels slides over the patch of image pixels

4.1 Convolution

Convolution is a mathematical process that involves multiplying a matrix (known as a filter or kernel) with an input image element or pixels to produce an output feature map. This technique is commonly used in Convolutional Neural Networks (CNNs) to extract useful information from images e.g. edges, texture, scenes, etc.Prior to performing convolution, it is essential to verify the dimensions of the input image to determine whether padding is required or not.

4.2 Padding

Padding is defined as the addition of layers of pixels to an image when it is being processed by the CNN kernel. Pixels are added to the outer frame of the image to allow more space for the filter to cover the image. It results in a more accurate analysis of images.

Types of padding:

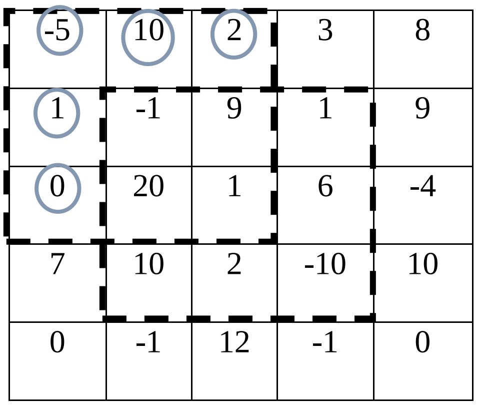
There are two types of padding:

1. Same Padding (It uses a stride of 1, and pixels will be added. The layer's outputs will have the same spatial dimensions as its inputs).
2. Valid Padding (There will be no stride or no addition of pixels in the input image. The layer only uses valid input data).

Why is padding necessary?

Padding is necessary because it helps reduce the loss of information at the borders of the input feature map and can improve the performance of the model. When the kernel moves over the image, it will move more in the center and will have more knowledge about the middle part of the image than the edges, and we might lose important features from the edges.

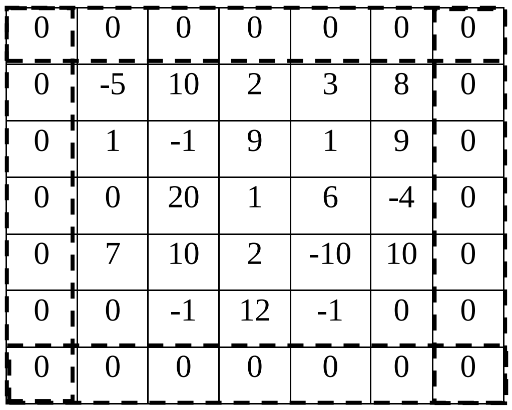
Suppose below this is an image matrix, and the black dashed line squares are kernels that will move over an image. As we can see from the below picture, the pixels at the edges (circled numbers) will be used one time, and there is a high chance we may lose a piece of information, whereas the pixels in the middle, enclosed in the second black dashed line square, will be used multiple times when performing MAC operations.



-5	10	2	3	8
1	-1	9	1	9
0	20	1	6	-4
7	10	2	-10	10
0	-1	12	-1	0

Figure 4.3: Lost of information due to MAC operation

The solution is to add more dimension according to the requirements of the image so that all the information that is at the edges of the image will not be lost. Now that this picture has two rows of zeros and two columns of zeros, this will help us get all the features of an image. The pixels that were at the edges are now in the middle, and when a filter is placed over them, we can easily detect the feature.



0	0	0	0	0	0	0
0	-5	10	2	3	8	0
0	1	-1	9	1	9	0
0	0	20	1	6	-4	0
0	7	10	2	-10	10	0
0	0	-1	12	-1	0	0
0	0	0	0	0	0	0

Figure 4.4: Padded rows and columns in an image

After the convolution is performed, the image size will reduce and the remaining operations will not perform correctly, so the padding will also help to have the same image size after the convolution.

How do we know that the image needs padding?

If the input image size is greater than the output image before padding, which means that the information is lost, then padding is necessary.

Formula to calculate the size of the image before padding:

Suppose the image is 5x5 and the filter is 3x3.

$$output\ image\ before\ padding = (m - f + 1)(n - f + 1) \qquad 4.1$$

To check if an image is getting the same output size as an input image after performing padding, we use the following formula:

$$output\ padded\ image = (m + 2p - f + 1)(n + 2p - f + 1) \qquad 4.2$$

m=number of rows, n=number of columns, f=filter dimensions, p= number of rows and columns padded

4.3 Architecture Description

4.3.1 Convolution Layer:

The convolution layer is the core building block of the CNN. It is responsible for performing the majority of the computations in the network [9].

Working of Convolution:

The filter, which is 3x3 in size, is like a small window that moves over the input image, taking one step at a time with a certain stride. It goes over every part of the image and multiplies the filter values with the corresponding pixel values of the image, resulting in the dot product calculation.

This process is also called the Multiplication and Accumulation (MAC) process, where the pixel values of the image are multiplied by the filter values and then added up together. After this operation, the output is stored in 2D. This operation is repeated until the entire image has been covered by the filter.

MAC Process:

$$
\begin{bmatrix} -5 & 10 & 2 & 3 & 8 \\ 1 & -1 & 9 & 1 & 9 \\ 0 & 20 & 1 & 6 & -4 \\ 7 & 10 & 2 & -10 & 10 \\ 0 & -1 & 12 & -1 & 0 \end{bmatrix}
\otimes
\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}
\longrightarrow
\begin{bmatrix} -56 & 30 & -26 \\ 131 & -29 & 30 \\ 32 & -21 & -106 \end{bmatrix}
$$

Figure 4.5: Operations of MAC process on an image matrix

(-5×-1)+(10×-1)+(2×-1)+(1×-1)+ (-1×8)+(9×-1)+(0×-1)+ (20×-1)+(1×-1)=-56

(-10×-1)+(2×-1)+(3 ×-1)+(-1×-1)+ (8×9)+(1×-1)+(20×-1)+ (1×-1)+(6×-1)=30

(2×-1)+(3×-1)+(8×-1)+(9×-1)+ (1×8)+(9×-1)+(1×-1)+ (6×-1)+(-4×-1)=-26

(-1×-1)+(-1×-1)+(9×-1)+(0×-1)+ (20×8)+(1×-1)+(7×-1)+ (10×-1)+(2×-1)=131

(-1×-1)+(9×-1)+(1×-1)+(20×-1)+ (1×8)+(6×-1)+(10×-1)+ (2×-1)+(10×-1)=-29

(9×-1)+(-1×-1)+(9×-1)+(1×-1)+ (6×8)+(-4×-1)+(2×-1)+ (-10×-1)+(10×-1)=30

(0×-1)+(20×-1)+(1×-1)+(7×-1)+ (10×-8)+(1×-1)+(7×-1)+ (10×-1)+(2×-1)=32

(20×-1)+(1×-1)+(6×-1)+(2×-1)+ (10×-8)+(10×-1)+(12×-1)+ (-1×-1)+(0×-1)=-106

To check the output of a convolved image with striding, we use the following formula:

$$Stride\ Output\ Image\ dimensions\text{:} \left(\frac{m+2p-f+1}{s+1}\right)\left(\frac{n+2p-f+1}{s+1}\right) \qquad 4.3$$

In convolution, there will always be one stride. While performing convolution, padding will also be performed. Our image will be checked to see whether the padding is needed or not.

4.3.2 Maxpooling Layer

A pooling layer is used to decrease the dimensions of the feature map produced by the convolution layer [2]. By reducing the dimensions, the memory and processing requirements in the following layers will be reduced. Maxpooling is a pooling technique that selects the highest value from the specific region of the feature map covered by the filter. As a result, the output from the maxpooling layer would be a feature map that highlights the most significant feature from the previous feature map.

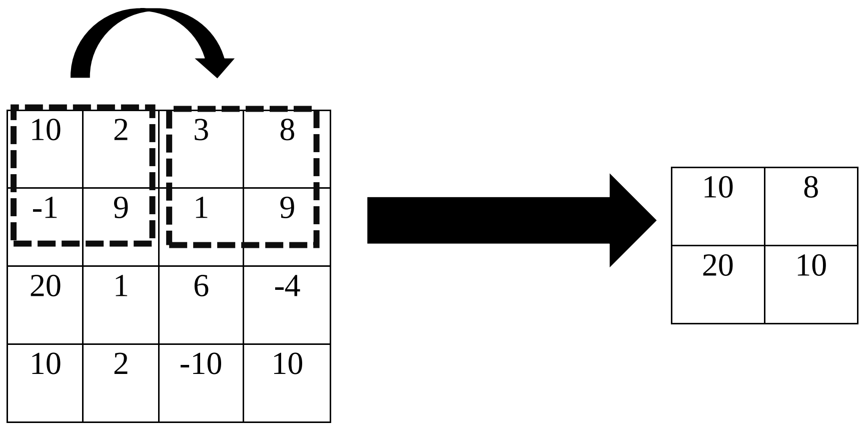


Figure 4.6: Process of Maxpooling on an image matrix

If the input image has an even m x n shape then there will be a stride of two. If an input image has an odd m x n shape, then there will be one stride.

4.3.2.1 Significance of Maxpooling:

Maxpooling is a technique used in neural networks to reduce the size of feature maps and prevent overfitting. It has several benefits, such as:

- Reducing the dimensionality of feature maps, which makes it easier and less expensive to process and store data. Additionally, max-pooling provides translation invariance, meaning that small translations in the input do not affect the output of the pooling operation. This is particularly useful in tasks such as object recognition, where the object's position in the image can vary.
- Another important advantage of max-pooling is that it can help prevent overfitting. By reducing the size of the feature maps, it forces the network to learn only the most important features, preventing it from iterating on the training data and allowing it to better generalize to new data.

- Maxpooling also introduces non-linearity into the network, which enables it to learn more complex relationships between inputs and outputs. Most real-world problems are non-linear, so this is an important feature of neural networks.
- Finally, max-pooling is a computationally efficient operation that can be performed quickly, making it easier to train deep neural networks with large amounts of data. Overall, max-pooling is a critical technique in neural networks that can improve the network's efficiency and performance, while also preventing overfitting and enhancing the network's ability to generalize to new data.

4.3.3 Fully Connected Layer

Fully connected layers convert a 2D array into a 1D array.

Flatten Expression:

$$result\_flat = input.flatten() \qquad 4.4$$

10	2	3	8
-1	9	1	9
20	1	6	-4
10	2	-10	10

$\Rightarrow$

10	2	3	8	-1	9	1	9	20	1	6	-4	10	2	-10	10

Figure 4.7: Conversion of 2D array into 1D array using flatten technique

4.3.3.1 Significance of a Fully Connected Layer:

Flattening layers help in reducing the dimensions of the input, which can help reduce the trainable parameters in the network, improve the computational efficiency, and make the neural network architecture less complex and simple.

4.3.4 Activation Functions

4.3.4.1 Rectified Linear Unit (ReLu):

The function of ReLu is to convert all the negative numbers into the positive numbers.

What does the activation function ReLu do?

The rectified linear activation function or ReLu is a short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

Why is ReLu most commonly used?

The main advantage of using the ReLu function over other activation functions is that it does not activate all the neurons at the same time and avoid saturation and vanishing gradient during training [8]. Another reason to employ the ReLu function is its ability to reduce computational costs by eliminating the need to compute exponentials in hardware, which is necessary when utilizing the sigmoid non-linearity function.

Expression of ReLu:

$$y\_relu = np.maximum(0, input) \tag{4.5}$$

10	2	3	8
-1	9	1	9
20	1	6	-4
10	2	-10	10

➡

10	2	3	8
0	9	1	9
20	1	6	0
10	2	0	10

Figure 4.8: Implementation of ReLu Activation Function

4.3.4.2 Significance of ReLu:

The ReLu (Rectified Linear Unit) activation function is frequently used in neural networks due to its several benefits.

- **Sparsity:**
  ReLu can set the output of a neuron to zero when the input is negative, which leads to sparsity in the network. This sparsity helps prevent overfitting by reducing the number of parameters that the model needs to learn.
- **Efficiency:**

ReLu is a simple function that is computationally efficient, meaning that networks that use ReLu can be trained faster and require less memory to store.

- **Non-Linearity:**
  ReLu has the ability to introduce non-linearity into the network. Most real-world problems are non-linear, and by introducing non-linearity, ReLu enables the network to learn more complex relationships between the inputs and the outputs.
- **Robustness to Vanishing Gradients:**
  ReLu is also robust to the issue of vanishing gradients, which can occur in deep neural networks and make them difficult to train. Vanishing gradients occur when the gradients become very small, and the network stops learning. ReLu can help mitigate this problem by preventing the gradients from becoming too small.

4.3.4.2 Sigmoid:

Sigmoid is a mathematical function used in neural networks to introduce non-linearity. This function is usually used for binary classification problems that take any real value as input and maps the output values in the range of 0 to 1.When the input value is more positive the output value will be closer to the 1.0, while when the input is more negative the output will be closer to 0.0.

Sigmoid Expression:

$$sig\_out = \frac{1}{(1+np.exp(-result\_flat))} \qquad 4.6$$

Sigmoid is also differentiable, which is important for gradient-based optimization methods used to train neural networks. Additionally, sigmoid can be used for normalization of the output of a neural network by scaling it to a range between 0 and 1. However, sigmoid is not always the best choice for an activation function. It can suffer from the vanishing gradient problem, where the gradients become very small as the input to the sigmoid function gets very large or very small, making it difficult to train deep neural networks using sigmoid.

Significance of Softmax:

The sigmoid function is a mathematical function that has a "S"-shaped curve. It is often used as an activation function in artificial neural networks. The sigmoid function has a number of properties that make it well-suited for this role.

- **Non-linearity:**
  The sigmoid function is non-linear, which means that it does not have a linear relationship with its inputs. This is important for neural networks because it allows them to learn complex relationships between their inputs and outputs.
- **Smoothness:**
  The sigmoid function is smooth, which means that it has no sharp edges. This is important for neural networks because it allows them to learn gradual changes in their inputs and outputs.
- **Output range:**
  The sigmoid function outputs values between 0 and 1, which is a convenient range for representing probabilities. This is important for neural networks because they are often used for classification tasks, where the goal is to predict the probability that an input belongs to a particular class.

4.3.4.3 Softmax:

Softmax is a mathematical function used in the last layer of neural networks for classification tasks. Its main significance is that it gives the output of the network as a probability distribution over predicted classes, making it useful for multiclass classification problems. Softmax normalizes the output of the network so that the sum of the predicted probabilities of all the classes is equal to one, which ensures that the predicted probabilities can be compared with different inputs easily.

Softmax Expression:

$$softmax = \frac{exp^{inputs}}{\sum exp^{inputs}} \tag{4.7}$$

Significance of Softmax:

The advantage of softmax is that it is differentiable, which means that we can calculate its gradient for the network's weights and biases. This is important for training the network using techniques such as gradient descent.

Softmax is a powerful activation function that allows us to train neural networks using standard optimization techniques and make accurate predictions on classification tasks.

Why is Sigmoid preferred over Softmax activation function?

When we build a Convolutional Neural Network (CNN), we have to decide which activation function to use in the last layer of the network. The choice between sigmoid and softmax depends on the task at hand and the type of data we are working with [1].

Sigmoid is typically used in binary classification problems, where the goal is to classify inputs into one of two possible outcomes (e.g., spam vs. not spam). The sigmoid function maps any input to a value between 0 and 1, which can be interpreted as the probability of the input belonging to the positive class. Implementing the sigmoid function in Verilog coding is relatively simple and can be done using just a few logic gates.

On the other hand, softmax is used in multiclass classification problems, where the goal is to classify inputs into one of several possible outcomes (e.g. cat, dog, or bird). The softmax function maps the inputs to a set of values that sum up to 1, which represent the probabilities of the inputs belonging to each class. Implementing softmax in hardware can be more complex than implementing sigmoid, as it involves more complex computations such as exponentiation and division.

5. Python System Model (Forward Propagation)

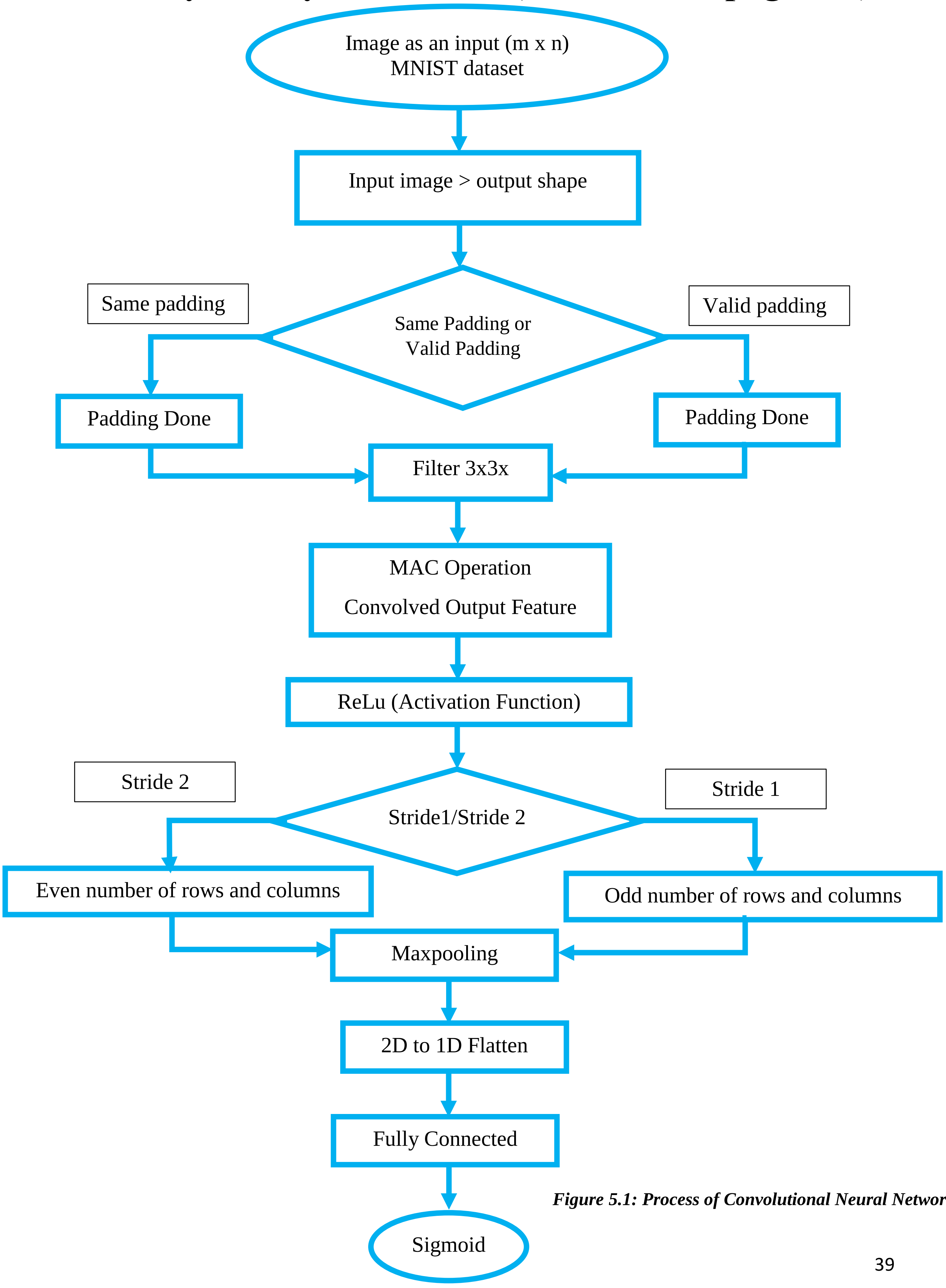


Figure 5.1: Process of Convolutional Neural Network

5.1 Process of Convolutional Python Model

This flow chart explains the overall workings of the Python model. The significance of this Python model is that it has helped us develop the hardware structure of convolutional neural networks on an FPGA. We have extracted the weights from the trained model and stored them in the memory.

5.1.1 Explanation of Process

Stage 1:

An image is loaded from a CSV file, and then it is processed. First, the shape of the input image and the expected output image are calculated and then compared to check if the size of the image is greater than the size of the output image or not.

Stage 2:

If the input image is larger than the output image, the program checks whether padding is needed or not. If padding is required (same padding), the program will prompt the user to enter the number of rows and columns they want to pad. The user can also specify the value of pixels they want to pad with, such as 0, 1, -1, or other values. If the input image size is smaller than or equal to the output image size, the program will not pad (valid pad) the image and move on to the next stage.

Stage 3:

In stage 3, we enter the values of filters and how many dimensions of a filter are required. When we are done with it, it displays the matrix of a filter. Here we are using a 3x3 filter.

The reasons for using the 3x3 filter over other filters, e.g., 5x5, 7x7, are as follows:

1. Firstly, it requires fewer parameters (weights) to learn, which makes it faster to train and requires less memory to store.
2. Secondly, a 3x3 filter can capture more fine-grained details in an image, such as edges and corners. This is important for image recognition tasks.
3. Thirdly, using a 3x3 filter allows for better translation invariance, which means the network can recognize the same feature in different parts of the image.

4. Lastly, a 3x3 filter can be used in a technique called depth-wise separable convolution, which reduces the number of parameters in a neural network while maintaining high accuracy.

Stage 4:

In stage 4 of the process, a convolution operation is performed on the image. A filter is moved over all pixels and performs the multiply and accumulate (MAC) process, storing the resulting values in a new matrix that will be our output feature map.

As the MNIST dataset consists of grayscale handwritten digits, each image has 2 channels. Therefore, our filter will cover these two channels. However, if we were using a 3-channel RGB picture, we would need 3 filters for each of the 3 channels, and the results would be combined at the end.

Stage 5:

The output feature map will then go to the layer of ReLu, where it will remove all the negative values (less than 0) by replacing them with 0 and keep all the positive values as they are. ReLu has a straightforward interpretation. When the input to a neuron is positive, the neuron fires, and when it is negative, the neuron does not fire. This makes it easier to understand what the network is doing and why it is making certain predictions.

Stage 6:

Maxpooling is a process in which a 2x2 window moves over the output feature map of the ReLu layer. The maximum value within the window is selected and stored in a new matrix. Since unnecessary pixels have already been removed from the image in the ReLu layer, this step further reduces the size of the image and the number of pixels.

The stride parameter is used to slide the window over the image. The code checks the dimensions of the image before selecting the stride value. If the image dimensions are even, the window moves with a stride of 2. If the dimensions are odd, the window moves with a stride of 1. This ensures that all parts of the image are covered by the window.

Stage 7:

In this stage, which is the flattening layer, it is used to convert the output of the previous layer into a one-dimensional array.

Stage 8:

This is the last stage and the last layer of the convolutional neural network where we see the probability of our output. The output with the highest probability will be the correct classification answer. Sigmoid is not a good choice for implementing in the last layer of CNN, but we have preferred it because it’s hardware is easier to design as compared to Softmax.

The choice also depends on the problems that are being addressed. For example, for binary classification problems, sigmoid is a good choice, while for multi-class classification problems, softmax is preferred.

6. Python System Model (Back Propagation)

Back Propagation plays a significant role in training the neural network. In this process the errors are calculated and transfer back to every layer where the weights parameters adjust their value according to error.

6.1 Back Propagation Introduction

Backpropagation is an important technique used in neural networks to train them for various tasks such as classification, regression, and other prediction problems. In convolutional neural networks (CNNs), backpropagation is used to adjust the weights of the network during the training phase.

CNNs are a type of neural network that is well suited for image processing tasks. They work by applying a series of filters to the input image, extracting the relevant features, and passing those features through the process to the fully connected layer, where it makes the prediction. During the training phase, the trainable parameters (weights) of the filter are fully connected and adjusted using backpropagation.

So far, we are following this flow. Now we are going to perform backward calculations, which will result in updating weights.

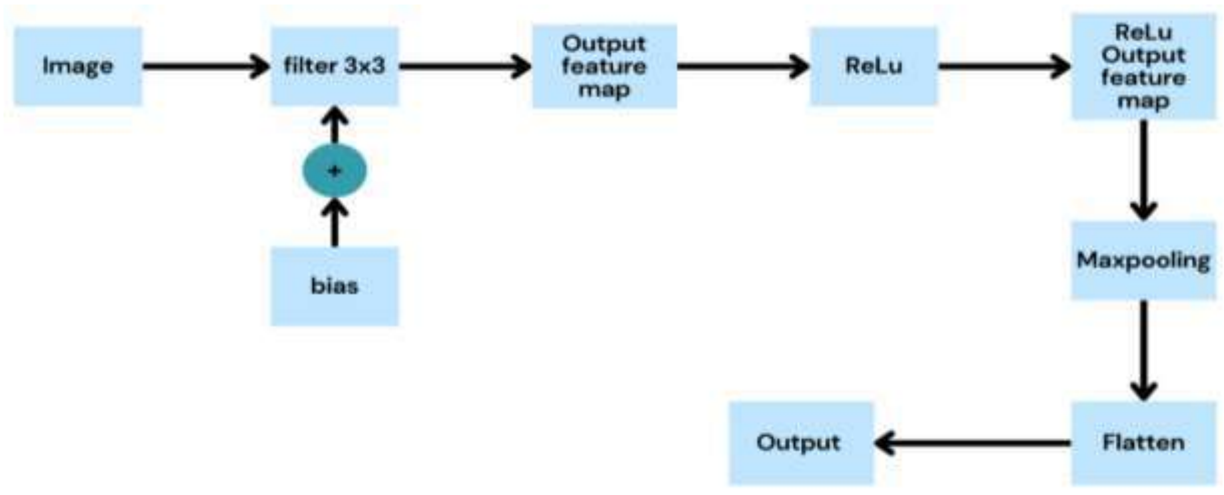


Figure 6.1: Block Diagram of back propagation in Python Model

Backpropagation minimizes the error between the predicted output and the actual output. It is a process to calculate the gradient loss of the function with respect to each weight in the network. The loss functions tell us how the network is performing on the given data, and the gradient of loss tells us how much weight needs to be adjusted.

Keeping figure-6.1 in mind, the logic flow chart is drawn:

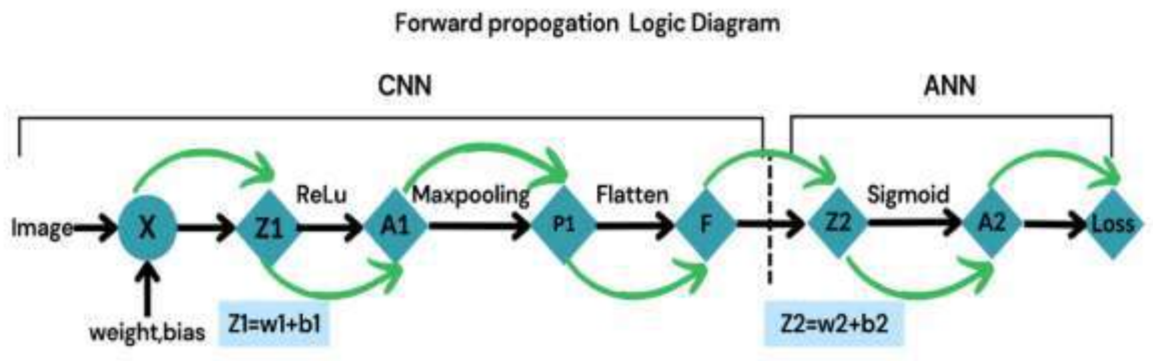


Figure 6.2: Logical Diagram of forward propagation in Python Model

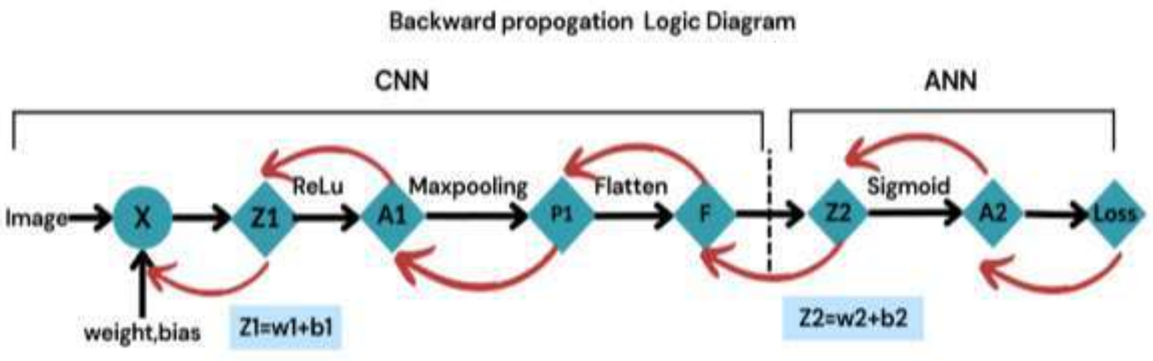


Figure 6.3: Logical Diagram of backward propagation in Python Model

The figures 6.2 and figure 6.3 shows that the change in every layer in forward and backward propagations, the change in parameters affects the next parameter, similarly, in the reverse direction, every error changes the parameter. For backpropagation, we use the chain rule.

6.2 Finding the Back Propagation Equation:

To find the equation and verify it, we will assume the dimension of a model

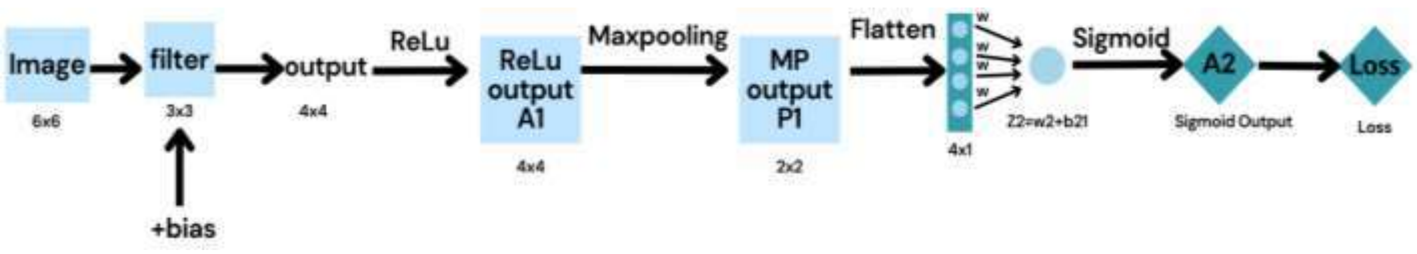


Figure 6.4: Simplest Logical Diagram Representation

Trainable parameter in the suppose model:

W1= (3, 3) W2= (1, 4) b1= (1, 1) b2= (1, 1)

Total 15 trainable parameters (suppose model)

Loss Function: (Binary Classification)

Loss to find for single image:

$$L = -Yi\, log\big(\hat{Y}i\big) - 1(1 - Yi)\, log(1 - \hat{Y}i) \qquad 6.1$$

Loss to find on the batch of image:

$$L = \frac{1}{m}(-Yi\, log\big(\hat{Y}i\big) - 1(1 - Yi)\, log(1 - \hat{Y}i)) \qquad 6.2$$

Our objective is to minimize loss values for w1, w2, b1, b2.

Gradient descent:

$$w1 = w1 - \eta\frac{\delta L}{\delta w1} \qquad w2 = w2 - \eta\frac{\delta L}{\delta w2} \qquad b1 = b1 - \eta\frac{\delta L}{\delta b1} \qquad b2 = b2 - \eta\frac{\delta L}{\delta b2}$$

6.3 Artificial Neural Network (ANN):

Starting from the ANN part. In figure figure-6.1 of the forward propagation logic diagram, in ANN portion, we have neurons from the fully connected layer that have trainable parameters. Z2 is showing the output of the weights and bias, which are coming from the previous layer then passing through the sigmoid which gives output A2, and then loss.

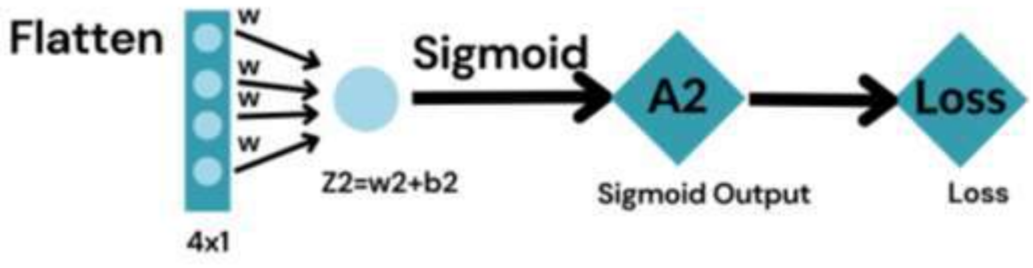


Figure 6.5: Block Diagram of Artificial Neural Networks (ANN)

6.3.1 Forward Propagation Equation of ANN:

$$Z_2 = w_2F + b_2 \tag{6.3}$$

$$A_2 = \delta(Z_2) \tag{6.4}$$

6.3.2 Backward Propagation Equation of ANN:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial w_2} \tag{6.5}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial b_2} \tag{6.6}$$

For backward propagation we take partial derivatives, but partial derivatives are not directly possible so chain rule is apply. It is applied because w2 and b2 are indirectly related to loss.

Let's take one single image to get the equations of $\frac{\partial L}{\partial w_2}$ then we can scale our calculations for more than one image.

Let's assume, a_2 is representing single image. It can be any image of any MNIST dataset.

$$\frac{\partial L}{\partial a_2} = \frac{\partial}{\partial a_2}[Y_i \log(a_2) - (1 - Y_i)\log(1 - a_2)]$$

$$\frac{\partial L}{\partial a_2} = \frac{-Y_i}{a_2} + \frac{(1 - Y_i)}{(1 - a_2)}$$

$$\frac{\partial L}{\partial a_2} = \frac{-Y_i(1 - Y_i) + a_2(1 - Y_i)}{(1 - a_2)}$$

$$\frac{\partial L}{\partial a_2} = \frac{-Y_i + Y_i a_2 + a_2 - Y_i a_2}{(1 - a_2)}$$

$$\boldsymbol{\frac{\partial L}{\partial a_2} = \frac{(a_2 - Y_i)}{a_2(1 - a_2)}}$$

$$\frac{\partial A_2}{\partial Z_2} = \delta(Z_2)[1 - \delta(Z_2)]$$

$$\boldsymbol{\frac{\partial A_2}{\partial Z_2} = a_2[1 - a_2]}$$

$$\frac{\partial Z_2}{\partial w_2} = \frac{\partial}{\partial w_2}(w_2 F + b_2)$$

$$\frac{\partial Z_2}{\partial w_2} = F + 0$$

$$\frac{\partial Z_2}{\partial w_2} = F \qquad \frac{\partial Z_2}{\partial b_2} = 0$$

$$\frac{\partial L}{\partial w_2} = \frac{a_2 - Y_i}{\cancel{a_2(1 - a_2)}} \times \cancel{a_2(1 - a_2)} \times F$$

$$\boldsymbol{\frac{\partial L}{\partial w_2} = (a_2 - Y_i) \times \mathrm{F}}$$

Replacing the a_2 (single image) with A_2 which is multiple image:

$$\boldsymbol{\frac{\partial L}{\partial w_2} = (A_2 - Y_i) \times F^T}$$

$$\frac{\partial L}{\partial b_2} = \frac{a_2 - Y_i}{\cancel{a_2(1-a_2)}} \times \cancel{a_2(1-a_2)} \times 1$$

$$\frac{\partial L}{\partial b_2} = = (\boldsymbol{A_2} - \boldsymbol{Y_i})$$

When we will work with multiple images ‘m’ ($\frac{1}{m}$ will be multiplied with formulas)

6.4 Convolutional Neural Network (CNN):

In the CNN part, we will find the backpropagation of the Maxpooling Layer and Convolution Layer. In the below picture, we have written all the equations that are derived from the logic diagram.

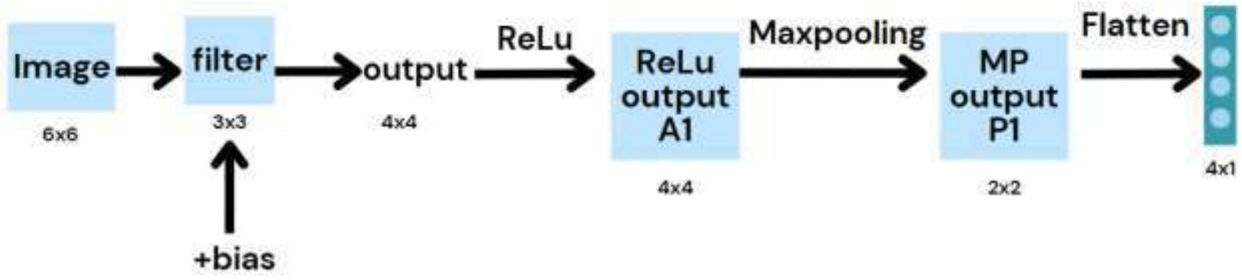


Figure 6.6: Logical Diagramm of CNN in an i mage using the Python Model

In the CNN part, we have w1 and b1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial F} \times \boxed{\frac{\partial F}{\partial P_1}} \times \boxed{\frac{\partial P_1}{\partial A_1}} \times \frac{\partial A_1}{\partial Z_1} \times \boxed{\frac{\partial Z_1}{\partial w_1}}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial F} \times \boxed{\frac{\partial F}{\partial P_1}} \times \boxed{\frac{\partial P_1}{\partial A_1}} \times \frac{\partial A_1}{\partial Z_1} \times \boxed{\frac{\partial Z_1}{\partial b_1}}$$

Flatten Back Propagation

Will be derive from back propagation of convolution

Pooling Back Propagation

6.4.1 Forward Propagation Equation of CNN:

$$Z_2 = w_2 F + b_2$$

$$A_2 = \delta(Z_2)$$

$A1 = relu(Z1)$ $\qquad$ $Z_1 = conv(x, w_1) + b_1$

$P_1 = maxpooling(A_1)$ $\qquad$ F=flatten (P_1)

$Z_2 = w_2 F + b_2$ $\qquad$ $A_2 = sigmoid(Z_2)$

We have to find:

$w1 = w1 - \eta \frac{\delta L}{\delta w1}$ $\qquad$ $b1 = b1 - \eta \frac{\delta L}{\delta b1}$

$$\frac{\partial L}{\partial w_1} = \boxed{\frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2}} \times \frac{\partial Z_2}{\partial F} \times \frac{\partial F}{\partial P_1} \times \frac{\partial P_1}{\partial A_1} \times \frac{\partial A_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_1}$$

$$\frac{\partial L}{\partial b_1} = \boxed{\frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2}} \times \frac{\partial Z_2}{\partial F} \times \frac{\partial F}{\partial P_1} \times \frac{\partial P_1}{\partial A_1} \times \frac{\partial A_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial b_1}$$

We have find out these values

$$\frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2} = A_2 - Y_i$$

$Z_2 = w_1 F + b_1$ $\qquad$ $\frac{\partial Z_1}{\partial F} = w_1$

As there are no trainable parameters here in flatten layer, instead of taking derivatives we just do the reverse operation on it.

$F = flatten(P_1)$ $\qquad$ $\frac{\partial F}{\partial P_1} = reshape(P_1)$ or reshape.($P_1.shape$)

$$\frac{\partial L}{\partial w_1} = \boxed{\frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2}} \times \boxed{\frac{\partial Z_2}{\partial F}} \times \boxed{\frac{\partial F}{\partial P_1}} \times \frac{\partial P_1}{\partial A_1} \times \frac{\partial A_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_1}$$

$$\frac{\partial L}{\partial b_1} = \underbrace{\boxed{\frac{\partial L}{\partial A_2} \times \frac{\partial A_2}{\partial Z_2}}}_{\boxed{A_2 - Y_i}} \times \underbrace{\boxed{\frac{\partial Z_2}{\partial F}}}_{\boxed{w_2}} \times \underbrace{\boxed{\frac{\partial F}{\partial P_1}}}_{\boxed{reshape(P_1) \text{ or reshape.(} P_1.shape)}} \times \frac{\partial P_1}{\partial A_1} \times \frac{\partial A_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial b_1}$$

6.5 Back Propagation on Maxpooling:

The matrix we got at P2 is 2x2 which has 4 elements, and they are errors. P2=$\begin{bmatrix} X1 & X2 \\ X3 & X4 \end{bmatrix}$ errors.

There are also no trainable parameters in the maxpooling layer, we will do the reverse operation.

$$A1 = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 15 & 16 \end{bmatrix} \xrightarrow{\boxed{\text{Maxpooling}}} P1 = \begin{bmatrix} 4 & 8 \\ 12 & 16 \end{bmatrix}$$

Suppose we have a 4x4 matrix, and on that matrix, we applied a 2x2 window of max-pooling operation that gives us a 2x2 output matrix. When we have to do backpropagation, we will do the reverse operation of the 2x2 matrix.

The maximum values that we will get are the errors, and the remaining values in A1 of 2x2 will not progress further. It means the error values will be replaced back to their original positions in matrix A1, but the other values will be replaced with zero.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 8 \\ 0 & 0 & 0 & 0 \\ 0 & 12 & 0 & 16 \end{bmatrix} \rightarrow \frac{\partial L}{\partial A1}$$

So, $\frac{\partial L}{\partial A1} = \begin{cases} \frac{\partial L}{\partial P_{xy}}, & if\ A_{mxn}\ is\ the\ maximum\ element \\ 0, & otherwise \end{cases}$

6.6 Rectified Linear Unit (ReLu):

Here we will find the equations that will give us the back propagations results if ReLu layer.

6.6.1 Forward Propagation on ReLu:

$$A1 = relu(Z1)$$

6.6.2 Backward Propagation on ReLu:

No trainable parameters, reverse operation.

$$\frac{\partial A1}{\partial Z1} = \begin{cases} 1, & if\ Z_{1xy} > 0 \\ 0, & if\ Z_{1xy} < 0 \end{cases}$$

6.7 Back Propagation on the Convolution Layer:

Consider X as an image. We have a trainable parameter in the convolution layer.

$$\begin{tikzcd} \mathrm{X} \arrow[r] & \otimes \arrow[r] & Z_1 \\ \boxed{\text{(3x3)}} & \boxed{\text{W1, b1 (2x2)}} \arrow[u] & \end{tikzcd}$$

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \otimes\ w = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \Rightarrow\ Z_1 = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

The equations of the Z1 matrix are:

$$Z_{11} = X_{11}W_{11} + X_{12}W_{12} + X_{21}W_{21} + X_{22}W_{22} + b_1$$

$$Z_{12} = X_{12}W_{11} + X_{13}W_{12} + X_{22}W_{21} + X_{23}W_{22} + b_1$$

$$Z_{23} = X_{21}W_{11} + X_{22}W_{12} + X_{31}W_{21} + X_{32}W_{22} + b_1$$

$$Z_{22} = X_{22}W_{11} + X_{23}W_{12} + X_{32}W_{21} + X_{33}W_{22} + b_1$$

These are the forward propagations of the convolution of X and W, which results in the Z matrix.

We have to look for the backward propagation matrix.So for backward propagation, $\frac{\partial L}{\partial Z1}$ the matrix is:

$$\frac{\partial L}{\partial Z_1} = \begin{bmatrix} \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} \\ \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} \end{bmatrix}$$

Firstly we will find for b1 :$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_1}\ x\ \frac{\partial Z_1}{\partial b_1}$. To find $\frac{\partial Z_1}{\partial b_1}$ differentiate Z_1 with respect to b1.

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_1} \times \frac{\partial Z_1}{\partial b_1}$$

$$\frac{\partial L}{\partial b_1} = \boxed{\frac{\partial L}{\partial Z_{11}}} \times \boxed{\frac{\partial Z_{11}}{\partial b_1}} + \boxed{\frac{\partial L}{\partial Z_{12}}} \times \boxed{\frac{\partial Z_{12}}{\partial b_1}} + \boxed{\frac{\partial L}{\partial Z_{21}}} \times \boxed{\frac{\partial Z_{21}}{\partial b_1}} + \boxed{\frac{\partial L}{\partial Z_{22}}} \times \boxed{\frac{\partial Z_{22}}{\partial b_1}}$$

The values in the square box are already in the matrix of $\frac{\partial L}{\partial Z_1}$.

We have to find out the values that are enclosed in the dashed square box. The partial derivative of Z_{11}, Z_{12},Z_{21}, $Z_{22}$with respect to b1 will be 1.

The equations will become:

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_{11}} + \frac{\partial L}{\partial Z_{12}} + \frac{\partial L}{\partial Z_{21}} + \frac{\partial L}{\partial Z_{22}}$$

$$\frac{\partial L}{\partial b_1} = sum(\frac{\partial L}{\partial Z_1})$$

Now we will find out $\frac{\partial L}{\partial w_1}$ with respect to w1 $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_1}$.

$$\frac{\partial L}{\partial w_1} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} \end{bmatrix} \qquad \frac{\partial L}{\partial Z_1} = \begin{bmatrix} \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} \\ \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} \end{bmatrix}$$

$$\begin{aligned}
\frac{\partial L}{\partial w_{11}} &= \boxed{\frac{\partial L}{\partial Z_{11}}} \times \frac{\partial Z_{11}}{\partial w_{11}} + \boxed{\frac{\partial L}{\partial Z_{12}}} \times \frac{\partial Z_{12}}{\partial w_{11}} + \boxed{\frac{\partial L}{\partial Z_{21}}} \times \frac{\partial Z_{21}}{\partial w_{11}} + \boxed{\frac{\partial L}{\partial Z_{22}}} \times \frac{\partial Z_{22}}{\partial w_{11}} \\
\frac{\partial L}{\partial w_{12}} &= \boxed{\frac{\partial L}{\partial Z_{11}}} \times \frac{\partial Z_{11}}{\partial w_{12}} + \boxed{\frac{\partial L}{\partial Z_{12}}} \times \frac{\partial Z_{12}}{\partial w_{12}} + \boxed{\frac{\partial L}{\partial Z_{21}}} \times \frac{\partial Z_{21}}{\partial w_{12}} + \boxed{\frac{\partial L}{\partial Z_{22}}} \times \frac{\partial Z_{22}}{\partial w_{12}} \\
\frac{\partial L}{\partial w_{21}} &= \boxed{\frac{\partial L}{\partial Z_{11}}} \times \frac{\partial Z_{11}}{\partial w_{21}} + \boxed{\frac{\partial L}{\partial Z_{12}}} \times \frac{\partial Z_{12}}{\partial w_{21}} + \boxed{\frac{\partial L}{\partial Z_{21}}} \times \frac{\partial Z_{21}}{\partial w_{21}} + \boxed{\frac{\partial L}{\partial Z_{22}}} \times \frac{\partial Z_{22}}{\partial w_{21}} \\
\frac{\partial L}{\partial w_{22}} &= \boxed{\frac{\partial L}{\partial Z_{11}}} \times \frac{\partial Z_{11}}{\partial w_{22}} + \boxed{\frac{\partial L}{\partial Z_{12}}} \times \frac{\partial Z_{12}}{\partial w_{22}} + \boxed{\frac{\partial L}{\partial Z_{21}}} \times \frac{\partial Z_{21}}{\partial w_{22}} + \boxed{\frac{\partial L}{\partial Z_{22}}} \times \frac{\partial Z_{22}}{\partial w_{22}}
\end{aligned}$$

The square box values are in the matrix $\frac{\partial L}{\partial Z_1}$. The values that are enclosed in the dashed box are what we have to find out.

Using Equation of $Z_{11}, Z_{12}, , Z_{21}$ and Z_{22} as previously described

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial Z_{11}} X_{11} + \frac{\partial L}{\partial Z_{12}} X_{12} + \frac{\partial L}{\partial Z_{21}} X_{21} + \frac{\partial L}{\partial Z_{22}} X_{22}$$

$$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial Z_{11}} X_{12} + \frac{\partial L}{\partial Z_{12}} X_{13} + \frac{\partial L}{\partial Z_{21}} X_{22} + \frac{\partial L}{\partial Z_{22}} X_{23}$$

$$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial Z_{11}} X_{21} + \frac{\partial L}{\partial Z_{12}} X_{22} + \frac{\partial L}{\partial Z_{21}} X_{31} + \frac{\partial L}{\partial Z_{22}} X_{32}$$

$$\frac{\partial L}{\partial w_{22}} = \frac{\partial L}{\partial Z_{11}} X_{22} + \frac{\partial L}{\partial Z_{12}} X_{23} + \frac{\partial L}{\partial Z_{21}} X_{32} + \frac{\partial L}{\partial Z_{22}} X_{33}$$

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} \quad \frac{\partial L}{\partial Z_1} = \begin{bmatrix} \frac{\partial L}{\partial Z_{11}} & \frac{\partial L}{\partial Z_{12}} \\ \frac{\partial L}{\partial Z_{21}} & \frac{\partial L}{\partial Z_{22}} \end{bmatrix}$$

6.7.1 Results of Back Propagation of Convolution Layer:

$$\boldsymbol{\frac{\partial L}{\partial w_1} = conv(input, \frac{\partial L}{\partial Z_1})}$$

$$\boldsymbol{\frac{\partial L}{\partial b_1} = sum(input, \frac{\partial L}{\partial Z_1})}$$

The image, which is input, is getting convolved with the partial derivative of $\frac{\partial L}{\partial Z_1}$.

7. Results of Python Model

- The figure 7.1 and 7.3 shows validation accuracy graph of the model on the validation set over the course of training. The figure 7.1 and 7.3 shows training accuracy graph of the model on the training set over the course of training.
- The figure 7.2 and 7.4 shows validation loss graph of the model on the validation set over the course of training. The figure 7.2 and 7.4 training loss graph of the model on the training set over the course of training.

🟦 Training 🟧 Validation

7.1 Python model using Keras and TensorFlow:

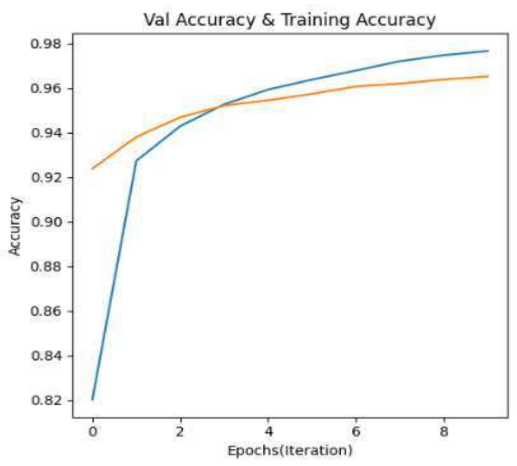


Figure 7.1: Validation Accuracy and Training Accuracy

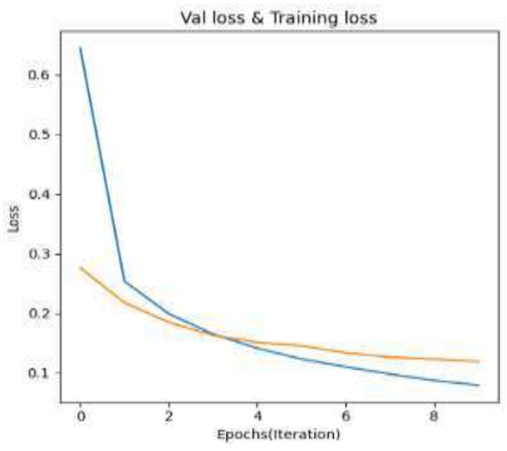


Figure 7.2: Validation Loss and Training Loss

7.2 Python Native Model:

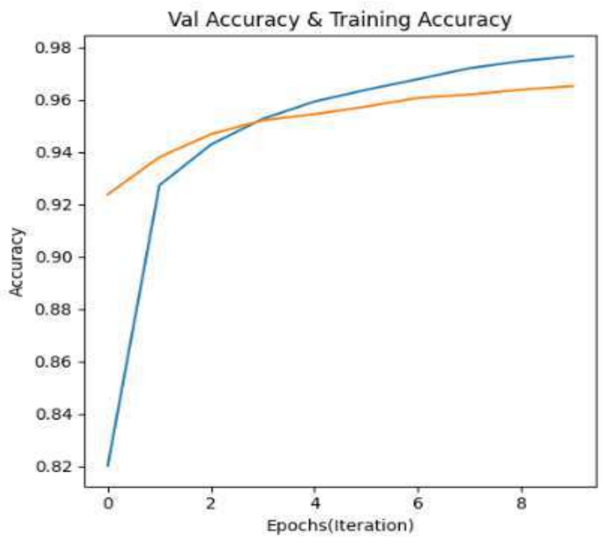


Figure7.3: Validation Accuracy and Training Accuracy

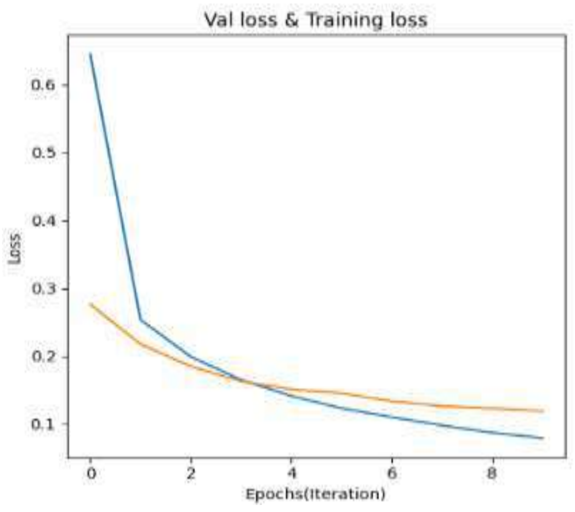


Figure 7.4: Validation Loss and Training Loss

8. Analysis of Results:

We want to see the validation accuracy and loss curves to be steadily increasing, and the training accuracy and loss curves to be steadily decreasing. This indicates that the model is learning and generalizing to new data as it is trained.

However, if the validation accuracy curve starts to plateau, this indicates that the model is overfitting to the training data. In this case, we can try to regularize the model, such as by adding dropout or L2 regularization.

If the validation loss curve starts to increase, this indicates that the model is not learning. In this case, we can try to increase the learning rate or the number of epochs.

9. Verilog Implementation of a CNN Accelerator

In applications involving image and signal processing, convolutional neural networks (CNNs) are a typical type of artificial neural network. CNNs, on the other hand, are computationally demanding and consume many resources.

Specialized hardware accelerators have been created to effectively complete the convolutional procedures needed to solve this problem. The main objective of our proposed accelerator is to target the low area and energy-efficient design of mobile/edge training hardware. The accelerator's training has been evaluated for MNIST handwritten digits 0–9.

9.1 Blocks of the CNN Accelerator:

A typical CNN accelerator employing Verilog, which is a hardware description language, can be easily divided into multiple parts.

The first part is the input buffer, which stores the input data that needs to be processed. The buffer is created to effectively store input data, which is often in the form of a two-dimensional matrix or image.

The accelerator's primary computational duty is then completed by the convolutional core. The processing components execute the process of convolution on the input data from the convolutional core. A single multiplication and addition process is carried out by each processing component, and the output can be obtained by combining the results.

The output buffer is made to efficiently manage the vast amounts of data generated by the convolutional core and stores all results generated by the convolutional operation.

To make the output data less complex and smaller in size, the pooling layer down samples it. There are several methods, like average pooling and maximum pooling that can be used to implement the pooling layer.

The CNN accelerator's final output is generated by the fully connected layer following a series of matrix multiplications.

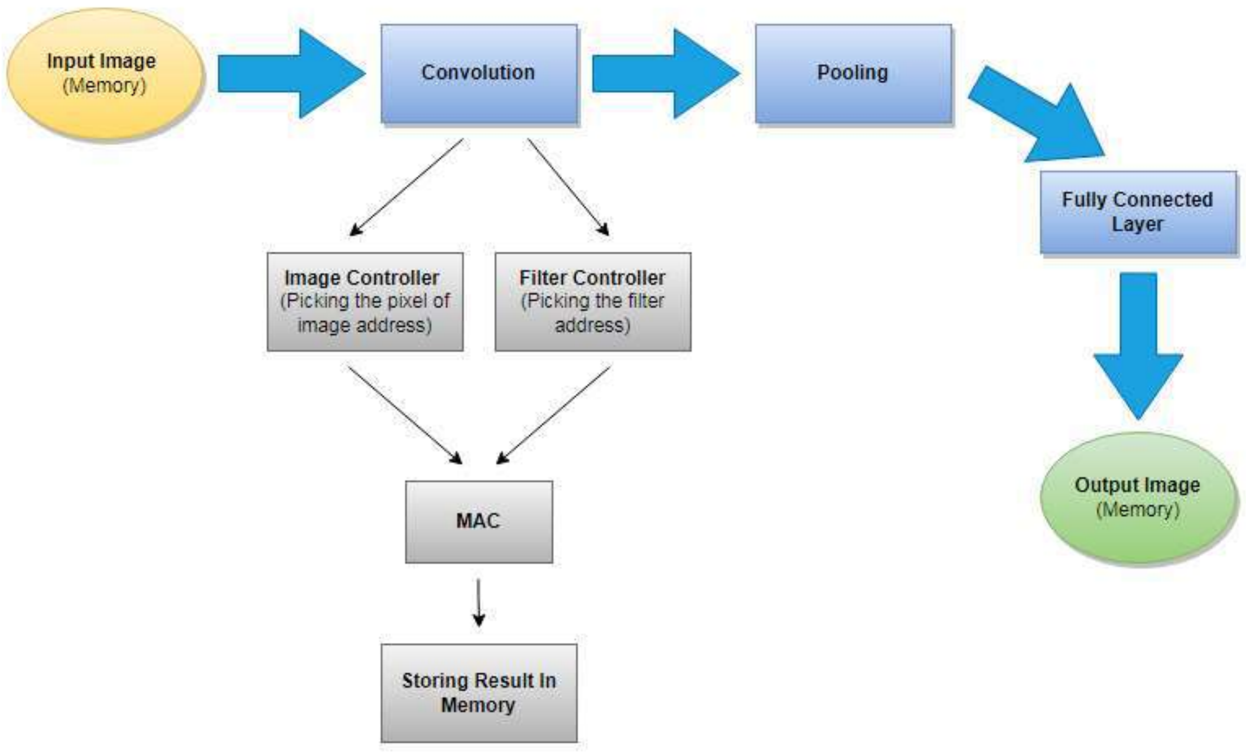


Figure9.1: Block Diagram of CNN with Verilog

Overall, the Verilog block diagram of a CNN accelerator gives a high-level understanding of the accelerator's design. Each component in the block diagram has properties and functionality that are described using the Verilog language. A hardware synthesis tool is then used to convert the Verilog code into real hardware.

9.2 Description of each block of the CNN Accelerator:

9.2.1 Convolution Layer:

A convolutional layer in CNN applies a series of learnable filters (also known as kernels) to the input data to extract features and learn spatial hierarchies. Concerning Verilog implementation, the following is a brief description of a convolutional layer:

A convolutional layer in Verilog can be developed by combining memory access, multiplication, and addition operations. The layer comprised various crucial parts, including:

1. **Input Buffer:** An input buffer stores the input data, such as an image. The input data is temporarily stored in this buffer, providing quick access throughout the convolution process.

2. **Convolution Operation:** The convolution operation is the primary calculation of a convolutional layer. To create feature maps, a series of filters must be applied to the input data. A matching patch of the input data is convolved with a small matrix of weights that comprise each filter.
    - **Weight Memory:** A weight memory is used to store the filter weights. The filter weights are stored in this memory, which is accessible throughout the convolution process.
    - **Filter Multiplication:** Using a set of multipliers, the input data is multiplied by the filter weights. The input data and the filter weights are multiplied element by element by these multipliers.
    - **Accumulation:** Adders are used to collect the multiplication results to provide the output values for the convolution process.
3. **Activation Function:** An activation function is frequently used to induce non-linearity after the convolution procedure. ReLu (Rectified Linear Unit) or sigmoid functions are frequently used as activation functions. Combinational logic can be used to achieve this phase in Verilog.
4. **Output Buffer:** An output buffer is used to hold the feature maps that are generated by the convolutional layer. This buffer stores the interim findings and enables quick access during the CNN's later layers or stages.

Convolutional layers are implemented in Verilog by building and connecting the above-mentioned parts. Pre-trained filter weights may be used to initialize the weight memory. The convolution procedure is carried out by cycling through the filters and executing the appropriate multiplications and additions as the input data streams through the layer. The generated feature maps are then stored in the output buffer after being processed by the activation function.

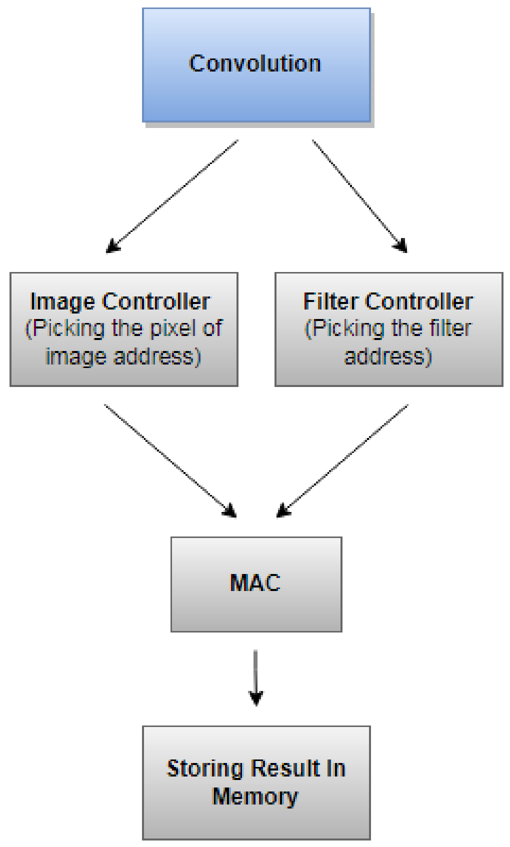

Convolution
Image Controller (Picking the pixel of image address)
Filter Controller (Picking the filter address)
MAC
Storing Result In Memory

Figure 9.2: In-depth block diagram of Convolution Layer in HDL

A convolutional layer implementation in a CNN might need complicated Verilog code, which is dependent on several different factors like filter sizes, data accuracy, parallelism, and memory organization. The implementation procedure may be made simpler, and the performance of the convolutional layer in Verilog can be improved with the support of libraries, IP cores, or high-level synthesis (HLS) tools provided by vendors like Xilinx or Intel.

9.2.2 Pooling Layer:

The spatial dimensions of the feature maps created by the convolutional layers are reduced by a pooling layer in a convolutional neural network (CNN). It helps in parameter reduction, feature map down sampling, and the introduction of a kind of translation invariance. A pooling layer in the context of a Verilog implementation is described briefly as follows:

Max pooling and average pooling are two methods that may be used to construct a pooling layer in Verilog. The layer is made up of the following key elements.

1. **Input Buffer:** An input buffer is used to hold the feature maps that the convolutional layer returns. The feature maps are temporarily stored in this buffer during the pooling process.
2. **Pooling Operation:** Applying a pooling operation to the input feature maps is the primary part of a pooling layer's computation.
    - **Pooling Window:** A fixed-size window that slides across the input feature maps is typically 2x2 or 3x3. The neighborhood of values across which the pooling operation is applied is defined by this window.
    - **Max Pooling:** The maximum value found within the pooling window is chosen as the representative value for that window when using the max pooling operation, for example. The most important characteristic in that area is captured with the aid of this operation.
3. **Output Buffer:** An output buffer is used to hold the down sampled feature maps that are part of the pooling layer's output. This buffer stores the interim findings and enables quick access during the CNN's subsequent layers or stages.

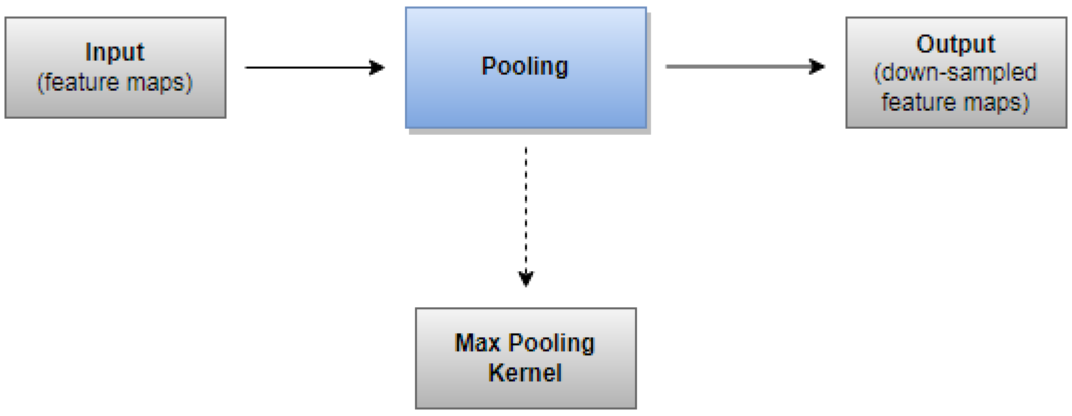


Figure 9.3: In-depth block diagram of Pooling Layer in HDL

The above-mentioned elements must be designed and connected to construct a pooling layer in Verilog. The pooling action is carried out by sliding the pooling window across the input feature maps as they are streamed through the layer. The output buffer is subsequently used to hold the resultant down sampled feature maps.

9.2.3 Fully Connected:

In a neural network, a fully connected layer, also referred to as a dense layer, links every neuron in the previous layer to every neuron in the current layer to generate a completely connected graph. In the context of Verilog implementation, the following is a brief description of a completely linked layer:

A completely linked layer may be created in Verilog by multiplying the matrix, adding the bias, and then applying the activation function. The layer is made up of the following key elements:

1. **Input Buffer:** An input buffer stores the input data from the previous layer. To temporarily store the input data while performing matrix multiplication, this buffer is used.
2. **Weight Memory:** A weight memory stores the weights used to link the neurons of the previous layer to those of the current layer. During the matrix multiplication process, this memory is accessed to obtain the weights.
3. **Matrix Multiplication:** The matrix multiplication operation is the primary calculation of a fully linked layer. The output values for each neuron in the current layer are created by multiplying each element of the input data by the corresponding weight and then adding the results.
    - **Weight Memory Access:** The connections between the neurons in the previous and current layers are used to access the weights from the weight memory.
    - **Element-wise Multiplication:** A set of multipliers is used to execute element-wise multiplication on the input data and corresponding weights.
    - **Accumulation:** Adders are used to collect the multiplication results in order to provide the output values for the fully linked layer.
4. **Bias Addition:** After the matrix has been multiplied, a bias term is added to the output value of each neuron. The bias values are applied to the output values using adders after being stored in a biased memory.
5. **Activation Function:** In order to add non-linearity, an activation function is often applied to the output values. ReLu (Rectified Linear Unit) or sigmoid functions are frequently used as activation functions.

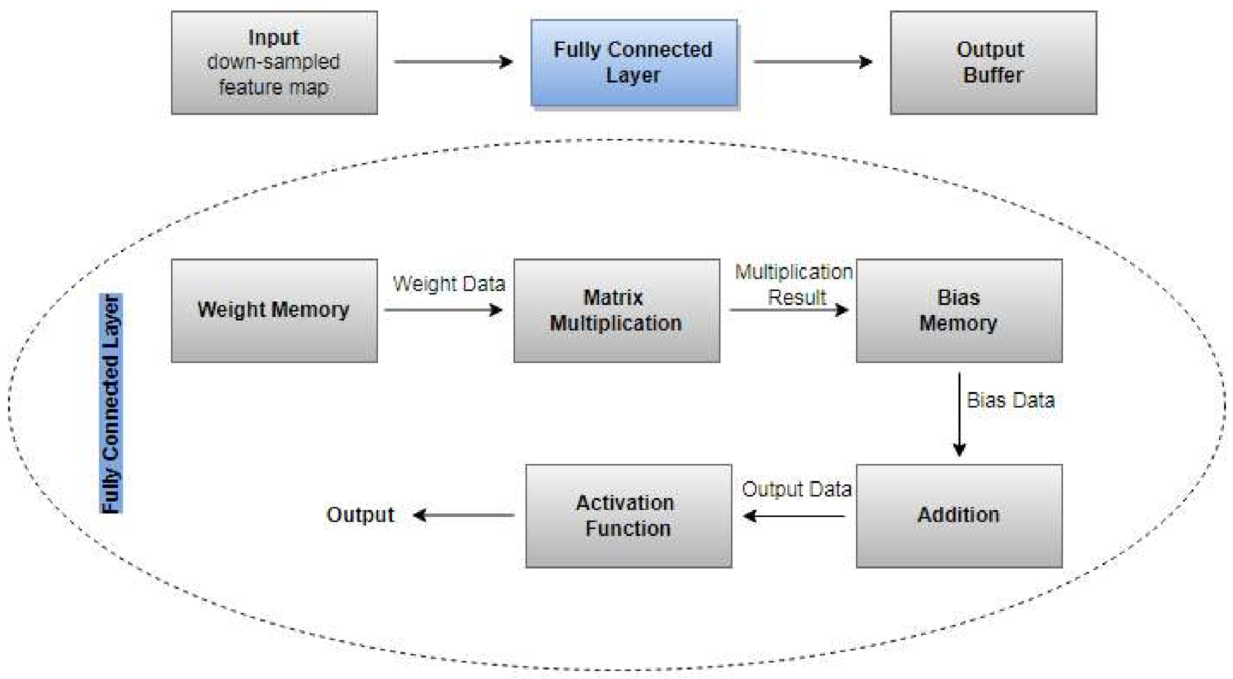
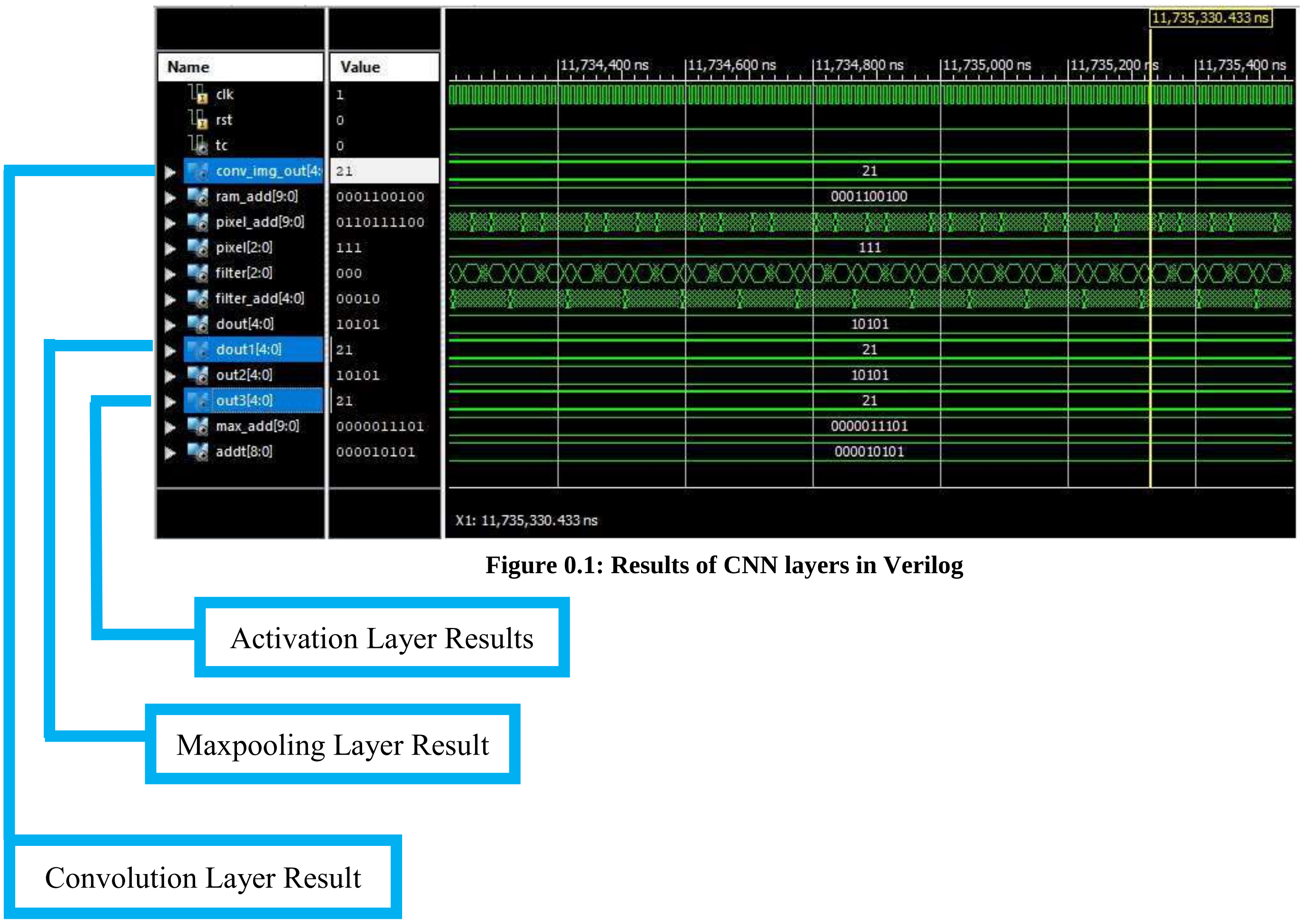


Figure 9.3: Block diagram of Fully Connected Layer in HDL

The Verilog implementation of a fully connected layer involves designing and interconnecting the components mentioned above. The input data is streamed through the layer, and the matrix multiplication operation is performed by cycling through the weights and applying the necessary multiplications and additions. The resulting output values are then passed through the bias addition and activation function stages.

10. Verilog Implementation of Convolutional Neural Network: Results



```
Name                 Value
clk                  1
rst                  0
tc                   0
conv_img_out[4:      21
ram_add[9:0]         0001100100
pixel_add[9:0]       0110111100
pixel[2:0]           111
filter[2:0]          000
filter_add[4:0]      00010
dout[4:0]            10101
dout1[4:0]           21
out2[4:0]            10101
out3[4:0]            21
max_add[9:0]         0000011101
addt[8:0]            000010101
```

Figure 0.1: Results of CNN layers in Verilog

11. Outcome

The size and power of the neural network model tend to grow as the number of layers does as well. So, we used a layer-pruning and weight-quantization-based network compression method. As we have used only a single convolutional layer, the CNN accelerator can significantly speed up computations and use less energy than conventional CPU-based solutions by optimizing each component for performance and efficiency.

We have developed an improved CNN as a result that is shallower, uses fewer filters, and has smaller bit widths for the parameters of weight while still being highly accurate. We picked a modest CNN along with the MNIST dataset for the training and validation, with the target accuracy set at 95% or above to demonstrate the viability of the suggested architecture and design methods.

12. Future Work

A few possible areas for future work, to either improve performance or extend functionality, are listed below:

12.1 Testing the Accelerator on Another Neural Network Model:

This future work has a potential research direction for evaluating the different neural network model in the developed ASIC based Neural Network Accelerator. By applying the accelerator to a different model, researchers can analyze its compatibility, efficiency, and overall performance in a wider range of deep learning applications. This future work helps to assess the generalizability and versatility of the accelerator design and provides insights into its effectiveness in different neural network architectures. It can also uncover any potential limitations or areas for improvement, guiding future iterations or modifications to enhance the accelerator's performance and adaptability.

12.2 Improvement in energy efficiency, reduce latency, reduce cost:

There is always a room for the improvement the performance and for optimizing the design.

- **Improve energy efficiency:**
  ASIC-based neural network accelerators can be made more energy efficient by using lower-power transistors and by optimizing the design for energy efficiency.
- **Support new neural network architectures:**
  ASIC-based neural network accelerators can be made to support new neural network architectures by using programmable logic or by using a heterogeneous architecture that combines ASICs with other types of processors.
- **Reduce cost:**
  ASIC-based neural network accelerators can be made to be more cost-effective by using standard cell libraries and by increasing the number of units that are produced.

As these technologies continue to develop, we can expect to see even more powerful and efficient ASIC-based neural network accelerators that will be used in a wide variety of applications.

12.3 Potential Applications of ASIC based Neural Network Accelerator:

Some of the applications are listed below:

12.3.1 Self-Driving Cars:

ASIC-based neural network accelerators could be used to power the computer vision systems that are used in self-driving cars. These systems need to be able to process large amounts of data in real time, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

12.3.2 Medical Imaging:

ASIC-based neural network accelerators could be used to power the systems that are used to analyze medical images, such as MRI scans and CT scans. These systems need to be able to process large amounts of data quickly, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

12.3.3 Virtual Reality:

ASIC-based neural network accelerators could be used to power the systems that are used to create virtual reality experiences. These systems need to be able to process large amounts of data in real time, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

12.3.4 Airborne Weapons:

ASIC-based neural network accelerators could be used to power the systems that are used to control and guide airborne weapons, such as missiles and drones. These systems need to be able to process large amounts of data quickly, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

12.3.5 Cancerous Cell Detections:

ASIC-based neural network accelerators could be used to power the systems that are used to detect cancerous cells in medical images, such as MRI scans and CT scans. These systems need to be able to process large amounts of data quickly, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

12.3.6 Telecommunications:

ASIC-based neural network accelerators could be used to power the systems that are used to transmit and receive data over telecommunications networks. These systems need to be able to process large amounts of data quickly, and ASIC-based neural network accelerators could provide the necessary performance and efficiency.

Bibliography

1. Machupalli, R., Hossain, M. and Mandal, M., 2022. Review of ASIC accelerators for deep neural network. *Microprocessors and Microsystems*, 89, p.104441.

2. Hong, J., Arslan, S., Lee, T. and Kim, H., 2021. Design of Power-Efficient Training Accelerator for Convolution Neural Networks. *Electronics*, 10(7), p.787.

3. Li, J., Un, K.-F., Yu, W.-H., Mak, P.-I., & Martins, R. P. (2021). An FPGA-based energy-efficient reconfigurable convolutional neural network accelerator for object recognition applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, *68*(9), 3143–3147. https://doi.org/10.1109/tcsii.2021.3095283

4. Ma, Y., Cao, Y., Vrudhula, S., & Seo, J.-S. (2020). Performance modeling for CNN inference accelerators on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *39*(4), 843–856. https://doi.org/10.1109/tcad.2019.2897634

5. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J., & Cao, Y. (2016). Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks. *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. https://doi.org/10.1145/2847263.2847276

6. Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., & Cong, J. (2017). Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. *Proceedings of the 54th Annual Design Automation Conference 2017*. https://doi.org/10.1145/3061639.3062207

7. K. He, et al., Deep residual learning for image recognition,in: Proceeding of the IEEE Conference on Computer Vision and Pattren Recognition, 2016

8. X.Glorot, A. Bordes, Y.Bengio,Deep sparse rectifier neural netwroks,in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics,2011.

9. Scanlan, A. G. (2018). Low power & mobile hardware accelerators for deep convolutional neural networks. *Integration*. https://doi.org/10.1016/j.vlsi.2018.11.010

10. Ševčík, P. (2016). A parallel FPGA implementation of image convolution. Master's thesis, Linköping University, Sweden.

11. Huang, W., Wu, H., Chen, Q., & Huang, Y. (2020). FPGA-based high-throughput CNN hardware accelerator with high computing resource utilization ratio. IEEE Transactions on Circuits and Systems II: Express Briefs, 66(12), 1500054. https://ieeexplore.ieee.org/document/8354237

12. Ao Ren, Ji Li., Zhe Li, & Caiwen Ding. (2018). Hardware SC-DCNN: Highly scalable deep convolutional neural network for FPGAs. IEEE Transactions on Very Large Scale Integration Systems, 26(1), 152-165. https://ieeexplore.ieee.org/document/8673634

13. Anthony G. Scanlan. A 40 Gbps CNN accelerator based on a high-performance FPGA architecture. Journal of Very Large Scale Integration Systems, 26(11), 1890-1905. https://ieeexplore.ieee.org/document/8354237

List of Acronyms

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
CNN	Convolutional Neural Network
DNN	Deep Neural Network
ANN	Artificial Neural Network
FC	Fully Connected