

Systolic Architecture Design for Convolutional Neural Networks in Low End FPGA

A Project Report

submitted by

Pranav T

*In partial fulfilment of the requirements
for the award of the degree of*

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

JUNE 2021

THESIS CERTIFICATE

This is to certify that the thesis **Systolic Architecture Design for Convolutional Neural Networks in Low End FPGA**, submitted by **Pranav T**, to the Indian Institute of Technology, Madras for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of the thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.



Dr K. SRIDHARAN

RESEARCH GUIDE

Professor

Dept. of Electrical Engineering
IIT MADRAS, 600036

Place: Chennai

Date: 16th June, 2021

ACKNOWLEDGEMENTS

It gives me great pleasure in expressing my sincere and heartfelt gratitude to my project guide Dr K Sridharan for his excellent guidance, motivation and constant support throughout my project. I consider myself extremely fortunate to have had a chance to work under his supervision. It has been a very learning and enjoyable experience to work under him.

My heartfelt appreciation to my friend Ahsan KV and Research scholar Yashraj Singh for spending their invaluable time with me in discussing about the project and answering my queries.

I would also like to thank my mother, who stood with me through all my tough times in my life. She truly is an amazing person. I would also like to extend my thanks to my dear friends Ajeel, Sandy, Rahul, Thira and Nandu for making the quarantined life bearable.

Finally, I would like to thank God almighty for His blessings due to which I was able to think in the right direction and complete the project successfully.

ABSTRACT

Character Image recognition task is identification of handwritten text by a computer. Deep learning techniques applied to character recognition has shown superior results in the past decade. FPGAs are a good platform to implement deep learning techniques because of its general purpose, flexibility and ability for reconfiguration.

Design and implementation of character recognition of a language on FPGA devices has numerous challenges to offer starting with obtaining good dataset, selection of algorithm, size of the network, computational cost and resources available in hardware device. In this project, the deep network is trained using a computer and the trained network is implanted in a low end FPGA using systolic architectures for character recognition.

This thesis examines the design of the digit recognition task on MNIST dataset and character recognition on Devanagari dataset. The overall tasks are grouped in three verticals – design of suitable training network, analysis of data quantization and design and implementation of a VLSI architecture to implement the task on a low end FPGA device.

A study was done to analyze different CNN architectures and identify a suitable deep network architecture first. The project moves on to implement the above architecture in Xilinx Spartan-3e FPGA by applying systolic architecture for the convolutions in the deep network.

CONTENTS

THESIS CERTIFICATE	i
ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
LIST OF FIGURES	vi
1 INTRODUCTION	1
1.1 Motivation.....	1
1.2 Objectives.....	2
1.3 Challenges	2
1.4 Contribution of this thesis.....	2
1.5 Organization.....	2
2 LITERATURE SURVEY.....	3
3 BASICS OF MACHINE LEARNING	6
3.1 Hypothesis function	6
3.2 Gradient Descent Algorithm	7
3.3 Activation functions	10
3.3.1 Sigmoid.....	10
3.3.2 Tanh	10
3.3.3 RELU	11
3.3.4 SoftMax.....	12
3.4 Improved Optimization algorithms.....	12
3.4.1 SGD.....	12
3.4.2 Minibatch Gradient descent	13
3.4.3 Momentum	13
3.4.4 RMSprop	14
3.4.5 Adam (Adaptive Momentum)	15
3.5 Deep learning network	15
3.5.1 Traditional approaches vs deep learning	16
3.5.2 Forward Propagation Equations	16
3.5.3 Backpropagation Equations	17
3.5.4 Weight update	18
3.5.5 Bias update.....	19

3.5.6	Previous layer update	20
3.6	Summary	21
4	OVERVIEW OF CONVOLUTIONAL NEURAL NETWORKS	22
4.1	Convolution operation	22
4.1.1	Edge Detection	23
4.2	Incorporating Convolutional layers in deep network	25
4.2.1	Forward Propagation	26
4.2.2	Activation functions	26
4.2.3	Backpropagation	27
4.2.4	Filter values update	30
4.2.5	Gradient to Previous Layer Evaluation	31
4.3	Pooling Layers	32
4.3.1	Max Pooling	33
4.3.2	Average Pooling	33
4.4	Strides in Convolution	37
4.4.1	Backpropagation with stride	38
4.4.2	Previous layer update	40
4.5	Padding in convolution	41
4.6	Summary	43
5	CHARACTER RECOGNITION USING NEURAL NETWORKS	44
5.1	Network with only Fully connected Layers	48
5.1.1	Shallow Network	49
5.1.2	FC1	49
5.1.3	FC2	51
5.1.4	FC3	52
5.1.5	FC4	53
5.2	Convolutional Layers only	55
5.2.1	CNN-1	55
5.2.2	CNN2	56
5.3	Comparison Report	58
5.4	Data Quantisation	59
5.5	Extension to Devanagari Dataset	63
5.5.1	Data Quantisation	65
5.6	Summary	66
6	VLSI ARCHITECTURE DESIGN FOR FPGA	67
6.1	Xilinx Spartan-3e FPGA Board	67

6.2	Storage Elements	67
6.2.1	Distributed RAM.....	67
6.2.2	Block RAM	68
6.2.3	Memory for CNN architecture	68
6.3	Overall Architecture.....	69
6.4	Non-systolic architecture for the Convolutional Layer	70
6.5	Systolic Architecture	71
6.5.1	2D convolution using systolic Design	72
6.6	Systolic Architecture for convolutional layer	74
6.6.1	Single PE	74
6.6.2	PE Array.....	75
6.6.3	RELU	77
6.7	Max Pooling	79
6.8	Max-finder Module	81
6.9	Experimental Results	81
6.9.1	MNIST data set.....	81
6.9.2	Devanagari character task	84
6.10	Summary	87
7	CONCLUSION AND FUTURE WORK.....	88
7.1	Contribution of the Thesis	88
7.2	Future Work and Extension	88
	REFERENCE.....	90

LIST OF FIGURES

Figure 1	A single layer neuron with an activation function.....	6
Figure 2	A sample 2nd order function	8
Figure 3	Initial point with $x_i > x_{min}$ (Slope marked at x_i with straight line).....	8
Figure 4	Initial point with $x_i < x_{min}$. (Slope marked at x_i with straight line)	8
Figure 5	change in convergence pattern with change in α	9

Figure 6 Sigmoid function and its derivative.....	10
Figure 7 tanh function and its derivative	11
Figure 8 RELU function and its derivative.....	11
Figure 9 (i) SGD without momentum (ii) with momentum.....	14
Figure 10 Traditional approach for classifications.....	16
Figure 11 multi-layer perceptron (MLP) for a multi class classification.....	16
Figure 12 Forward propagation for MLP.....	17
Figure 13 Backpropagation for a single neuron.....	18
Figure 14 A sample 6x6 image and its visual representation.....	23
Figure 15 A sample 2- layer CNN with activation functions.....	26
Figure 16 Back propagating cost functions.....	27
Figure 17 Back propagating bias and its updating.....	29
Figure 18 A sample 2-layer CNN to demonstrate number of computations.....	33
Figure 19 A sample 2-layer CNN to demonstrate number of computations after pooling	34
Figure 20 Backpropagation with average pooling and maxpooling	37
Figure 21 A sample 2-layer CNN	37
Figure 22 Backpropagation with stride.....	41
Figure 23 Backpropagation with padding.....	42
Figure 24 Sample of MNIST data set	44
Figure 25 Configuration of LeNet architecture.....	47
Figure 26 Accuracy and Loss function for the LeNet architecture	48
Figure 27 Accuracy and Loss function for the Shallow architecture.....	49
Figure 28 Configuration of FC1 architecture.....	50
Figure 29 Accuracy and Loss function for the FC1 architecture	50
Figure 30 Configuration of FC2 architecture.....	51
Figure 31 Accuracy and Loss function for the FC2 architecture	51
Figure 32 Configuration of FC3 architecture.....	52
Figure 33 Accuracy and Loss function for the FC3 architecture	53
Figure 34 Configuration of FC4 architecture.....	54
Figure 35 Accuracy and Loss function for the FC4 architecture	54
Figure 36 Configuration of CNN-1 architecture	55
Figure 37 Accuracy and Loss function for the CNN-1 architecture	56
Figure 38 Configuration of CNN-2 architecture	57
Figure 39 Accuracy and Loss function for the CNN-2 architecture	57
Figure 40 Some of the wrongly predicted images in CNN-2.....	59
Figure 41 : Histogram of weights and biases of CNN-2.....	60
Figure 42 Histogram of activations of 100 random images of each layer	61
Figure 43 Histogram of activations of 100 random images of each layer after scaling.....	62
Figure 44 Sample of Devanagari data set	63
Figure 45 Configuration of Devanagari architecture	64
Figure 46 Accuracy and loss function for Devanagari architecture.....	64
Figure 47 Schematic of input feature memory synthesized as dual port RAM	68
Figure 48 Single BRAM in memory module.....	69
Figure 49 Resource allocation	69
Figure 50 Overall implemented architecture in FPGA	69
Figure 51 A Non systolic architecture for convolution.....	71
Figure 52 Systolic architecture PE structure.....	72
Figure 53 Function of a simple PE.....	72

Figure 54 Dataflow in 2D convolution using systolic architecture.....	73
Figure 55 Single PE module	74
Figure 56 Resource utilization for single PE	75
Figure 57 conv PE array module.....	75
Figure 58 Resource utilization of one convolution PE array module	77
Figure 59 RELU+ Truncate components	78
Figure 60 Block diagram for convolutional layer	79
Figure 61 Block diagram for Maxpool layer	80
Figure 62 Maxpool module.....	80
Figure 63 Maxpool module resource utilisation	81
Figure 64 Block diagram for Max finder	81
Figure 65 Prediction result for digit 7	82
Figure 66 Prediction result for digit 2	82
Figure 67 Resource utilization with shifting in address generation.	82
Figure 68 Hardware with shifting used in address generation	83
Figure 69 Resource utilization with small multipliers used in address generation	83
Figure 70 Hardware utilization with small multipliers used in address generation	84
Figure 71 Prediction result for Devanagari dataset.....	85
Figure 72 Resource utilization for Devanagari character recognition using 12 bit	86
Figure 73 Resource utilization of Devanagari recognition with 10-bit quantization.....	86

LIST OF TABLES

Table 1 Comparison report of different Deep networks	58
Table 2 The accuracy values for various scale factors.....	62
Table 3 The accuracy values for various scale factors in 3.7 format	65
Table 4 The accuracy values for various scale factors in 3.9 format	66
Table 5 Scheduling of the inputs inside the PE.....	76

1 INTRODUCTION

Deep learning techniques are the go-to method in the field of Image recognition owing to its capability in achieving high accuracy compared to the traditional methods. The Neural network automatically learns the parameter values necessary to accurately predict the correct output. The latest neural network architectures have given an accuracy of over 99% in the fields of character recognition.

1.1 Motivation

Due to the computation pattern of the deep learning techniques, there is a large scope for parallel computations which makes GPUs, ASICs and FPGAs good platforms over a general-purpose CPU. FPGAs are especially good because of its good performance, ability for reconfiguration and massive parallelism. It also has the advantage of fast round development and finding its performance very easily.

Even though deep learning methods have significant advantages in terms of performance, it has its disadvantages in terms of requirements of heavy computational resources, need for huge memory to store the intermediate results and coefficient terms. This in turn requires large hardware and high-power consumption.

Even though the modern VLSI technologies has made possible to accommodate billions of components in a chip, effective utilisation of the available resources are needed to reduce the computation time and power consumption. This is particularly needed in implementing deep learning techniques in low end FPGAs where the availability of memory and hardware resources are less.

Because of the re-configurability of a FPGA, custom architecture which can utilise the maximum available resources and same time achieving maximum speed is required based on the custom application.

Systolic architecture is one of the architectures used for Google's Tensor Processor units (TPUs) which is an accelerator used for neural network machine learning using Tensor Flow software. This architecture uses a set of interconnected general-purpose cells, each performing a simple task with a high throughput. The general-purpose cells used are a set of Multiply and

Cumulate (MAC) units with an appropriate dataflow. Such MAC units can be easily replicated in FPGAs and the scope of such multiple MAC units operating in parallel can be determined based on the FPGA resources.

1.2 Objectives

This project focuses on identifying a CNN architecture keeping in mind the hardware limitations of the low-end FPGA (Xilinx Spartan 3e XC3S500) and implementing a VLSI architecture for the forward propagation computations for handwriting recognition.

1.3 Challenges

Identifying a proper CNN architecture is crucial as the filter sizes, in-channels, and out-channels determine the accuracy of the trained network. Increasing these numbers generally tend to increase the accuracy before reaching the over fitting problem. It also increases the number of intermediate features extracted and might lead to a large memory requirement as well as the need for bigger processing units.

1.4 Contribution of this thesis

CNN architecture for MNIST digit and Devanagari Characters Dataset for hardware limited devices.

Data quantization strategy for CNN with fixed point data.

VLSI architecture for character and digit recognition for low end FPGA device.

1.5 Organization

In the next chapter, fundamental building for machine learning algorithms and multi-layer perceptron model are presented.

In chapter 4 convolutional neural network basics and their algorithms are presented with numerical example.

In chapter 5 digit and character recognition architectures for MNIST dataset and Devanagari dataset and quantization strategy for fixed data points are presented along with experimental results.

In chapter 6 VLSI architecture for Low end FPGA devices and design considerations of different components are presented along with experimental results.

2 LITERATURE SURVEY

Deep learning techniques in the field of image recognition, natural language processing and speech recognition has given the best solutions in the past decade. The computer learns from the observational data automatically imparting a level of intelligence to the system.

The literature provided in [1] talks about the comprehensive understanding behind the image recognition of handwritten digits using neural networks. It also provides information on the core algorithms like back propagation, gradient descent, a visual proof on how a neural network can estimate any function, types of non-linearity and different layers involved in the network. The literature in [2] compares the different non linearity such as RELU, sigmoid and tanh and shows how that using RELU for supervised learning of deep networks is faster.

The literature provided in [3] talks about how the convolutional networks can be used for isolated character recognition. It also provides an architecture which showed an error rate of 0.7%. It also compares the deep learning technique results with other classifiers like SVM, k-Nearest neighbour, Principal component analysis etc. on a standard data set. This paper shows that better pattern recognition systems are based on automatic learning than the hand-crafted methods.

The literature provided in [4] increases the understanding on why CNNs performs so well. A visual technique to understand the functions of intermediate layers is implemented. These studies were done on the ImageNet database with Deep Net architecture. It also showed how these intermediate activations can be used to choose better architectures and obtain better results.

The literature provided in [5] provides an architecture which showed the best results on ImageNet database which has over 15 million labelled images in 22000 categories. The architecture implemented achieved the top-5 error rates of 15.3% significantly higher than the previous state of the art techniques. A recently new technique called dropout was implemented to reduce the overfitting in fully connected layers.

The literature in [6] improves the above architecture with the help of deeper networks (16-19 weight layers) with very small filters. All the convolutional filter was of size 3x3. It was also shown that this architecture generalises well to other data set with state-of-the-art accuracy.

The literature in [7], [8] looks into the performance of the deep networks where the weights and inputs are binary values. The performance of the system was only in percentages of mid 50s for the architecture in [5]. But the operations were 58 time faster and the memory savings of 32 times was achieved. This can be mostly employed in very low-end hardware.

The handwritten Devanagari character recognition was studied in [9], [10], [11]. There are some characters which look very similar to each other and the dataset available is not extensively used as in case of MNIST and ImageNet datasets. Lack of a good training data along with above issue makes it difficult to achieve accuracy compared to MNIST dataset. The maximum accuracy seen from the previous works is 98.26%.

Efficient hardware implementations of deep learning algorithms have been addressed in literatures.

There are literatures which talks about efficient Hardware accelerators with FPGA, which can be interphased with host device to accelerate deep learning algorithms. The literature in [12] put forward modified CNN architecture caffe with FPGA support which adapts the original GPU based architecture to FPGAs for parallel computations. It enables to achieves 50 GFLOPs performance boost. The literatures provided in [13], [14], [15] talks about optimization strategies of CNN accelerators and the levels of parallelism possible in convolution. Literature in [15] also discusses efficient data quantization strategies that can be performed for fixed bit width designs. It demonstrates the ways to reduce bit width without significant loss of accuracy. Literature in [16] combines flexibility of high-level synthesis and finer level optimization of RTL implementations to construct modularized and scalable RTL design of CNN algorithms. These are useful for reconfigurable CNN accelerators with FPGA devices of good resource limit.

Different types dataflow pattern is also discussed in literatures. Literature in [17] and [18] proposes energy efficient dataflow structures by minimizing memory access. Literature in [17] also provides analysing architecture of energy efficiency for different dataflow models. Literature [19] addresses the mismatch between parallel types supported by processing device and parallel type CNN workload has as dominant. It proposes a flexible dataflow model which optimizes interconnects in the dataflow and hence the resource utilization. The significance of interconnects in resource utilization for large number of processing elements is also demonstrated in this literature. The proposed model achieves 2-10 times boost in performance and 2.5-10 times improvement in efficiency.

Different types of architectures are also explored in literatures to speed up convolution time and increase efficiency³. Literature in [20] discusses implementation of 2D convolution on a chip using systolic architectures. Literature in [21] talks about what systolic array intends to do and design methods of systolic architectures. It also explains how systolic architectures helps to balance memory and computation in compute bound tasks.

Literature provided in [22] proposes a multiplier less architecture for CNNs. It enables the multiplication to be replaced with cordic as weights can be represented with trigonometric functions. It also talks about systolic architecture with limited number of processing elements along with data scheduling. For resource limited devices the architecture gives comparable performance.

Implementing fixed point is much more efficient than floating point on FPGA. So, most of the literatures use fixed point quantization. Different quantization strategies have also been discussed in literatures. Literature in [23] discusses different data quantization strategies applied to state of art CNN architectures. Literature in [24] observes that with data quantization of 20 bits almost zero accuracy loss is achieved for LeNet architecture.

3 BASICS OF MACHINE LEARNING

Any system having the capability to simulate human intelligence is said to have Artificial intelligence (AI). Machine learning is the form of AI that enables computers/systems to learn from a data rather than explicit programming. Similar to how human beings can be trained to learn things, computers are provided with a training data and a training network which models upon the training data to predict the desired output. The learned model can be deployed for the application. The learning can be supervised or unsupervised. In supervised learning, data are provided along with labels/expected output which the system has to learn and predict. E.g., Image classification where each image data is labelled with its corresponding class. In unsupervised learning only data is provided and no labels/expected output. An example is a clustering model which groups the customer database into different categories based on their common interest. Supervised models can be further classified into Regression models and Classification models. In regression models, output being predicted is a continuous quantity whereas in classification models, output being predicted is a discrete label. A model which predicts House rent in an area based on the data trained is a regression model while the model which predicts if a person is cancer patient or not is classification model.

A classification model can be binary or multiclass depending upon the number of classes it needs to classify. In classification problem, training data will be provided with labels and the model has to find the dependency of output on input data.

3.1 Hypothesis function

Consider a simple binary classification problem with one-layer network.

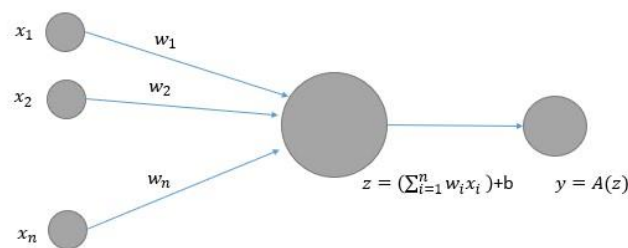


Figure 1 A single layer neuron with an activation function

Let $x_1, x_2, x_3, \dots, x_n$ be the input fed into model, z is the predicted output and y_0 is the label or expected output. Since classification is binary y_0 can be given as 0 and 1. Assuming dependency on all input values z can be written as

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

Where $w_1, w_2, w_3, \dots, w_n$ are weights each input contributing to the output and b is bias/offset value. Since the dependency need not be linear, A nonlinear element has to be included. They are called activation functions. The type of activation function to be chosen depends on the prediction problem. In the above case, since y has to predict the value to 0 or 1, function chosen must have range of 0 to 1. Then the predicted value will be the probability of being in class 1. Sigmoid function is a non-linear function which ranges from 0 to 1 and hence can be used.

$$\text{Sigmoid}(z) = A(z) = \frac{1}{1+e^{-z}}$$

Then y can be written as

$$y = A(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

So here the problem reduces to a function optimization problem where parameters $w_1, w_2, w_3, \dots, w_n$ and b has to be optimised so as to match y to labelled data y_0 . In order to solve this optimization problem, A Loss function/cost function $C(y, y_0)$ has to be associated. The Loss function which is a function of y and y_0 has very high value when difference between y and y_0 increases and zero/minimum when $y = y_0$. It basically reflects how much close each prediction value is to the expected output. Then the problem can be solved by optimizing the cost function with all training data and finding the optimized weights and bias values.

One such loss function is Mean Squared error (MSE) loss or L2 loss. It is defined as

$$C(y, y_0) = \frac{1}{2}(y - y_0)^2$$

3.2 Gradient Descent Algorithm

We have seen that if we can associate a loss function to a model, then the goal reduces to an optimization problem where loss function has to be minimized. One of the commonly followed approach for optimization is gradient descent approach. Convergence process of gradient descent algorithm and impact of hyperparameters in convergence are discussed below. These understanding can help to analysis the loss function while training the network model and tune the hyperparameters effectively.

Consider a function $y = f(x)$ as shown below.

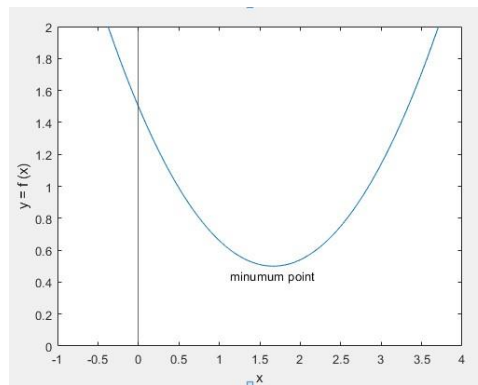


Figure 2 A sample 2nd order function

In order to find the minimum value for x ,

$x_{i+1} = x_i - \alpha f'(x_i)$ is iteratively computed.

If $x_i > x_{min}$, then $f'(x_i)$ is positive and x will decrease.

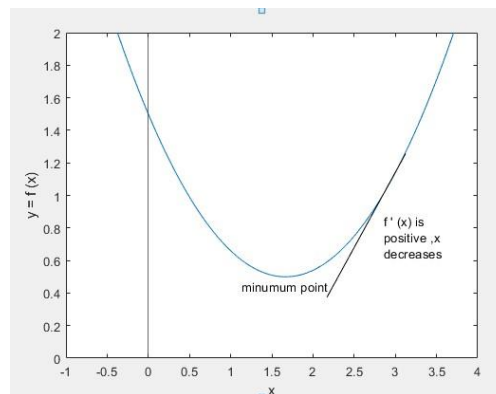


Figure 3 Initial point with $x_i > x_{min}$ (Slope marked at x_i with straight line)

If $x_i < x_{min}$, then $f'(x_i)$ is negative and x will increase.

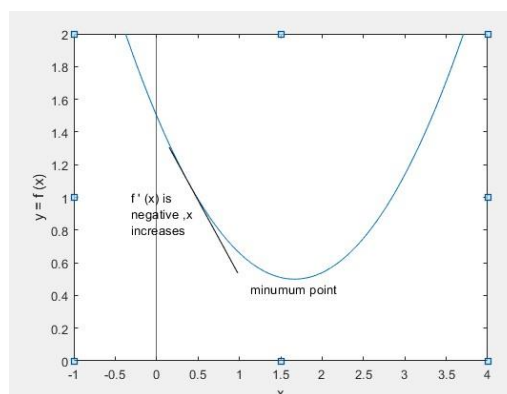


Figure 4 Initial point with $x_i < x_{min}$. (Slope marked at x_i with straight line)

α is a hyperparameter which is decisive while optimizing. If α is very large then Algorithm won't converge to minimum value. If α is too small it will take huge time to converge. Change in convergence pattern with increase in α is represented graphically below.

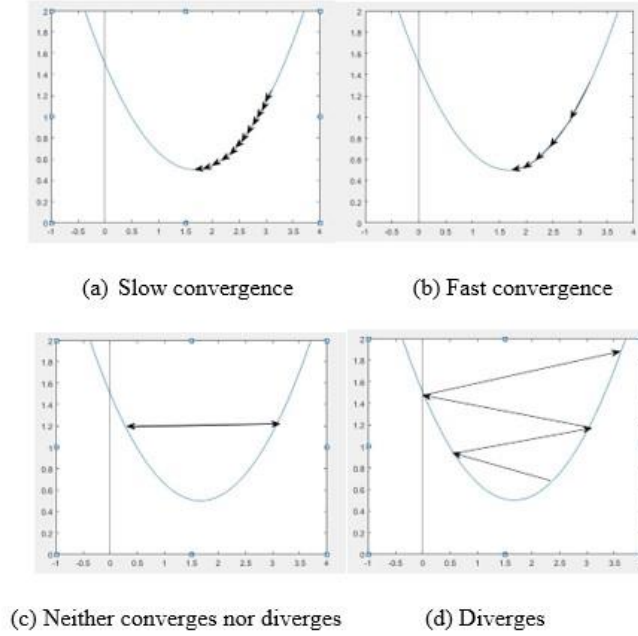


Figure 5 change in convergence pattern with change in α

Since the cost function has all the weights and biases as parameters, it is a function of several variables. In such functions, minimum value can be obtained if moved along the direction of the gradient ∇ (directional derivative), because function decreases at the maximum rate along the gradient. So, each weight is updated with corresponding partial derivative multiplied by step size α .

In the above example, if there are m training data and $y^{[k]}$ and $y_0^{[k]}$ represents k^{th} sample predicted output and expected output respectively, then cost function

$$C = \frac{1}{2m} \sum_{k=1}^m (y^{[k]} - y_0^{[k]})^2$$

Gradient component values,

$$\frac{\partial C}{\partial w_i} = \frac{1}{2m} \sum_{k=1}^m \left(\frac{\partial C}{\partial y^{[k]}} \cdot \frac{\partial y^{[k]}}{\partial z^{[k]}} \cdot \frac{\partial z^{[k]}}{\partial w_i} \right) \quad \frac{\partial C}{\partial y^{[k]}} = 2(y^{[k]} - y_0^{[k]}) \quad \text{and} \quad \frac{\partial z^{[k]}}{\partial w_i} = x_i^{[k]}$$

$$\text{Therefore } \frac{\partial C}{\partial w_i} = \frac{1}{m} \sum_{k=1}^m ((y^{[k]} - y_0^{[k]}) \cdot A'(z^{[k]}) \cdot x_i^{[k]})$$

$$\frac{\partial C}{\partial b} = \frac{1}{m} \sum_{k=1}^m \left(\frac{\partial C}{\partial y^{[k]}} \cdot \frac{\partial y^{[k]}}{\partial z^{[k]}} \cdot \frac{\partial z^{[k]}}{\partial b} \right) \quad \frac{\partial z^{[k]}}{\partial b} = 1$$

Therefore $\frac{\partial C}{\partial b} = \frac{1}{m} \sum_{k=1}^m ((y^{[k]} - y_0^{[k]}) \cdot A'(z^{[k]}))$

Then weights and biases can be updated as per equation below.

$$w_i' = w_i - \alpha \frac{\partial C}{\partial w_i}$$

$$b_i' = b_i - \alpha \frac{\partial C}{\partial b_i}$$

3.3 Activation functions

We have seen the optimization algorithm used in machine learning and its gradient update equations. Activation functions are another important aspect which is used in machine learning to bring non linearity in the network. Different types of activations used and their impact in introducing non linearity is discussed below. Fundamental Differences between them are also examined to get better idea of where these activations can be used.

3.3.1 Sigmoid

Sigmoid function is defined by the equation

$$f(x) = \frac{1}{1+e^{-x}} \quad f'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

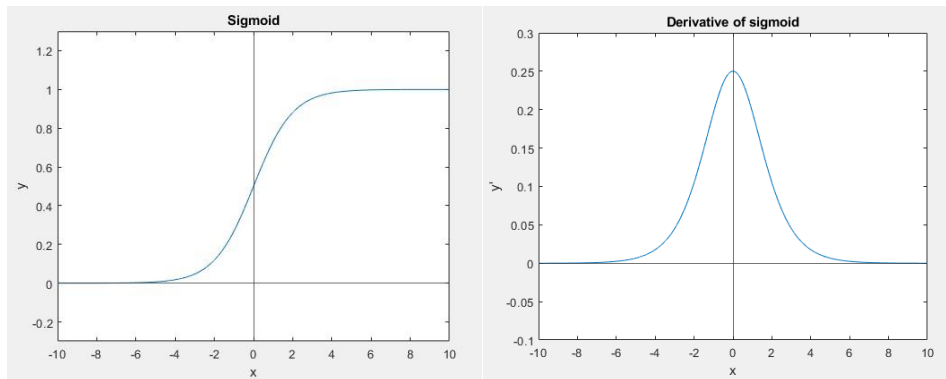


Figure 6 Sigmoid function and its derivative

It is a smooth function which is differentiable at every point. The function has a nice saturation at 1, limiting upper bound of any function. It is used mainly in output layer of binary classification models where output predicts the value between 0 and 1.

3.3.2 Tanh

Hyperbolic function Tanh is defined as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad f'(x) = \frac{4}{(e^x + e^{-x})^2}$$

It is easy to observe that tanh is a shifted form of sigmoid.

$$\tanh(x) = 2 \operatorname{sigmoid}(2x) - 1$$

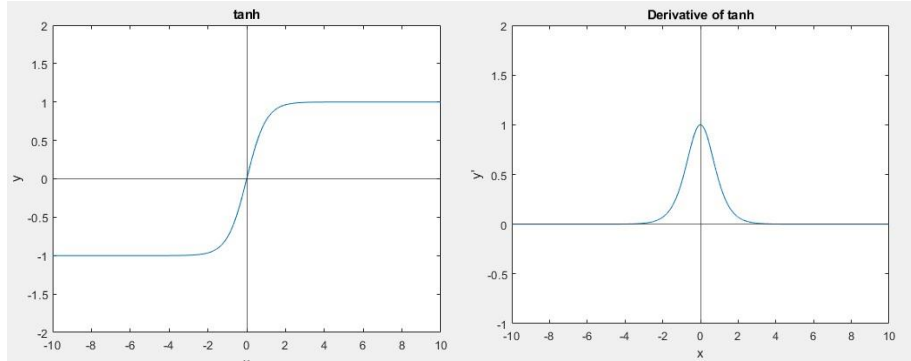


Figure 7 tanh function and its derivative

Since tanh is zero centered function, it is preferred over sigmoid in hidden layers as activation. Both sigmoid and tanh are computationally expensive to evaluate. Since the gradient of these functions at either end are close to zero, it sometimes causes ‘vanishing gradient’ problem. In the backpropagation algorithm, partial derivatives of the cost function backpropagate layer by layer. Zero value of the gradients of the activation functions make the derivative of the cost function to become zero while using chain rule. In that case network refuses to learn further. This is vanishing gradient problem.

3.3.3 RELU

Rectified Linear unit (RELU) is defined as

$$f(x) = \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \quad f'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

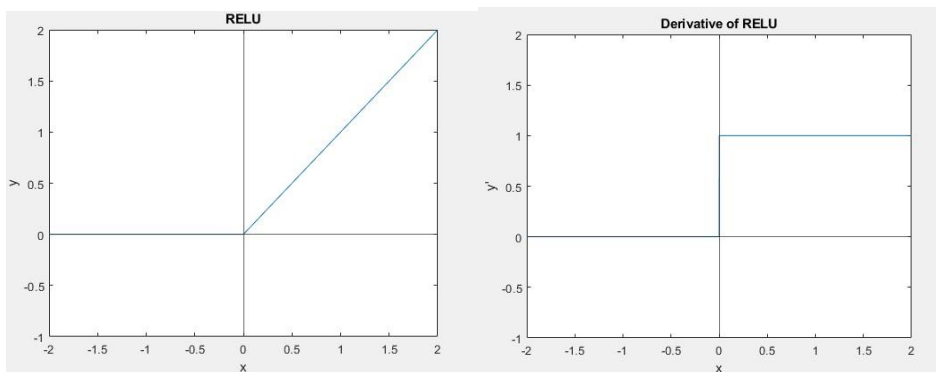


Figure 8 RELU function and its derivative

RELU helps to get rid of vanishing gradient problem. Also, it is computationally less expensive function. Hence it is commonly used activation in hidden layers.

3.3.4 SoftMax

SoftMax is used in multiclass classification problems where final output should be probabilities of each class which adds up to 1. If $x_1, x_2, x_3, \dots, x_n$ are values at each node of output layer before activation, then SoftMax function is defined as

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad \frac{\partial f}{\partial x_j} = \begin{cases} -\frac{e^{x_i} \cdot e^{x_j}}{(\sum_j e^{x_j})^2} & \text{if } j \neq i \\ \frac{e^{x_i}}{\sum_j e^{x_j}} - \left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right)^2 & \text{if } j = i \end{cases}$$

3.4 Improved Optimization algorithms

Apart from Batch gradient descent method discussed above some other modified algorithms are also used to arrive at optimized weights and bias values of network. the gradient descent equation discussed before uses all the training data points to evaluate cost function and hence gradient terms for weight update in each iteration. Due to high computational cost in this method, it is hardly used for big training data. Modifications to batch gradient descent algorithms and their update equations are discussed below. The improvement in convergence speed with these modifications and intuitive reasoning for the same is examined. Basic understanding of how each algorithms work can be useful in choosing the algorithm for optimization.

3.4.1 SGD

In stochastic gradient descent (SGD) method only one randomly sampled training data is used in each iteration to evaluate gradient and to update weights. Hence computational cost is reduced significantly with this method. it may not follow in the exact direction of minima and hence can take a longer time to converge. But increased no of iterations required is well compensated by the computational cost due to large no of samples.

For k^{th} randomly sampled data

$$C = \frac{1}{2} (y^{[k]} - y_0^{[k]})^2$$

Hence Weight and bias update equation will be

$$w_i' = w_i - \alpha (y^{[k]} - y_0^{[k]}) \cdot A'(z^{[k]}) \cdot x_i^{[k]}$$

$$b_i' = b_i - \alpha (y^{[k]} - y_0^{[k]}) \cdot A'(z^{[k]})$$

However Gradient direction oscillates due to the additional noise introduced by random selection of data in SGD.

3.4.2 Minibatch Gradient descent

It is a compromised method of optimization which reduces the high computational cost of Batch Gradient descent and oscillating nature of SGD. Here a minibatch of training data is used in each iteration for gradient evaluation and weight update.

$$C = \frac{1}{2m} \sum_{k=i}^{i+m} (y^{[k]} - y_0^{[k]})^2$$

Where C is the cost function for i^{th} minibatch and 'm' is the minibatch size.

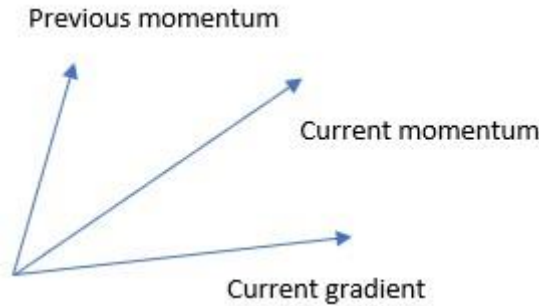
3.4.3 Momentum

Oscillating nature of SGD slows down learning. Momentum method helps to decrease runtime of convergence. Momentum term, which is exponentially weighted moving average of previous gradients is added to current gradient and parameters are moved along the resultant direction.

$$m(t) = \beta \cdot m(t-1) + (1 - \beta) \cdot g(t)$$

$$w(t+1) = w(t) - \alpha \cdot m(t)$$

Where β is a hyperparameter (set as 0.9 usually), $m(t-1)$ is momentum of previous iteration, $g(t)$ is current gradient, $m(t)$ is momentum in current iteration and $w(t)$ is weight in current iteration.



Let $\beta = 0.9$ and $m(0) = 0$

$$m(1) = (0.1)g(1)$$

$$m(2) = (0.9)(0.1)g(1) + (0.1)g(2)$$

$$m(3) = (0.9)^2(0.1)g(1) + (0.9)(0.1)g(2) + (0.1)g(3)$$

$$m(4) = (0.9)^3(0.1)g(1) + (0.9)^2(0.1)g(2) + (0.9)(0.1)g(3) + (0.1)g(4)$$

and so on. In general,

$$\begin{aligned} m(t) &= (1 - \beta)g(t) + (1 - \beta) \cdot \sum_{k=1}^{t-1} \beta^{t-k} \cdot g(k) \\ &= (1 - \beta)g(t) + \beta \cdot \frac{\sum_{k=1}^{t-1} \beta^{t-k} \cdot g(k)}{\sum_{k=1}^{\infty} \beta^{t-k}} \end{aligned}$$

Comparing with earlier equation,

$$m(t - 1) = \frac{\sum_{k=1}^{t-1} \beta^{t-k} \cdot g(k)}{\sum_{k=1}^{\infty} \beta^{t-k}}$$

It is evident that more weightage is given to most recent gradient terms than older terms in the momentum calculation.

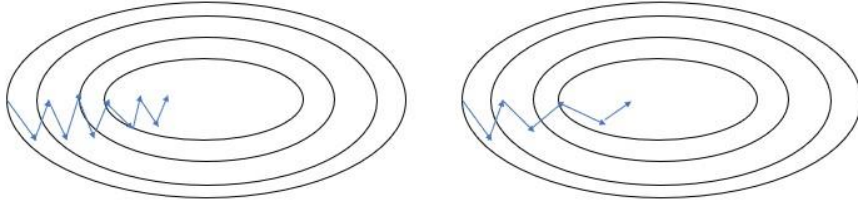


Figure 9 (i) SGD without momentum (ii) with momentum

momentum drives to direct along the actual minima and reduce oscillations in SGD as depicted above.

3.4.4 RMSprop

It uses rms value of gradient terms over the iterations and learning rate is adapted according to that for each parameter. Using single global learning rate for all weights makes convergence slower along the weights whose gradient is smaller compared to other weights. So, weights with smaller average gradient have to be updated with larger learning rate.

$$v(t) = \beta \cdot v(t - 1) + (1 - \beta) \cdot g^2(t)$$

$$w(t + 1) = w(t) - \frac{\alpha}{\sqrt{v(t)} + \varepsilon} \cdot g(t)$$

Where $v(t)$ is the exponentially weighted average of squares of gradient terms, ε is a very small constant added to avoid division by zero (typically given $\varepsilon = 10^{-8}$)

Global learning rate α is divided by rms value of gradient terms to get the actual learning rate for each parameter.

3.4.5 Adam (Adaptive Momentum)

It uses Momentum and rms prop together in the algorithm. Adaptive learning rate along with momentum terms are used while optimizing with bias correction terms also added.

$$m(t) = \beta_1 \cdot m(t-1) + (1 - \beta_1) \cdot g(t)$$

$$v(t) = \beta_2 \cdot v(t-1) + (1 - \beta_2) \cdot g^2(t)$$

$$\hat{m}(t) = \frac{m(t)}{1 - \beta_1^t}$$

$$\hat{v}(t) = \frac{v(t)}{1 - \beta_2^t}$$

Where β_1, β_2 are hyperparameters and $\hat{m}(t), \hat{v}(t)$ are bias corrected terms.

$$w(t+1) = w(t) - \frac{\alpha}{\sqrt{\hat{v}(t) + \varepsilon}} \cdot \hat{m}(t)$$

Bias correction is added in Adam because, momentum calculated (when initialized with zero momentum) uses $\sum_{k=1}^{\infty} \beta^{t-k}$ as sum of weights to find exponentially weighted average. To correct that bias, average is taken with $\sum_{k=1}^t \beta^{t-k}$ as sum of weights.

$$\frac{\sum_{k=1}^{\infty} \beta^{t-k}}{\sum_{k=1}^t \beta^{t-k}} = \frac{1}{1 - \beta_1^t}$$

Hence it is divided by term $1 - \beta_1^t$.

3.5 Deep learning network

Different optimization algorithms and Activation functions were discussed in preceding sections. In the following section we will see how cascading of multiple layers to form deep learning networks are used to get good modelling on the data. The working of forward and backward propagation equations of optimization algorithms with multiple layers are also examined.

3.5.1 Traditional approaches vs deep learning

In traditional approaches, handcrafted feature extraction techniques are used to extract features from the input data and those features are given as inputs to trainable classifiers to get output.



Figure 10 Traditional approach for classifications

In deep learning, multiple layers of abstractions are present in network itself and input data is given directly without any manual feature extractions. A set of nodes called neurons which are connected together to form an artificial neural network (ANN) which has biological inspiration from human neural system is used in deep learning. Cascade of nonlinear functions gives ANN power of universal function approximator, which enable them to realize any arbitrary mapping.

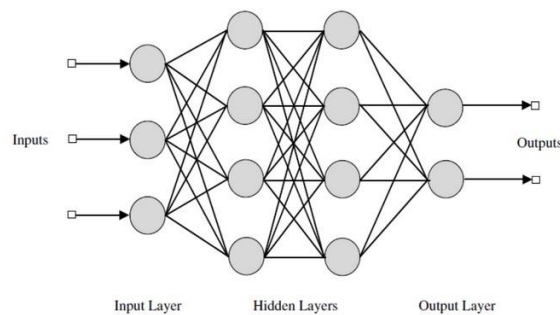


Figure 11 multi-layer perceptron (MLP) for a multi class classification

3.5.2 Forward Propagation Equations

Forward propagation equations are those which are involved in final output value and loss evaluation. It propagates from input layer to output layer. During inference phase, only these equations are used since optimization is not required. Forward propagation equations in multi-layer network is described below.

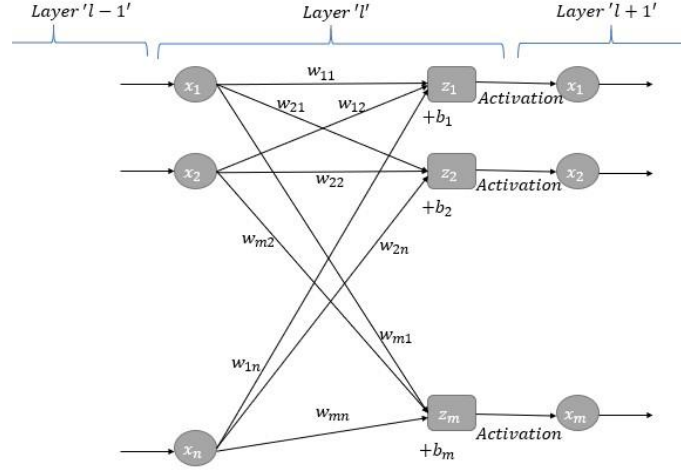


Figure 12 Forward propagation for MLP

For each layer 'l',

$$z_1^{[l]} = w_{11}^{[l]} x_1^{[l]} + w_{12}^{[l]} x_2^{[l]} + \dots + w_{1n}^{[l]} x_n^{[l]} + b_1^{[l]}$$

$$z_2^{[l]} = w_{21}^{[l]} x_1^{[l]} + w_{22}^{[l]} x_2^{[l]} + \dots + w_{2n}^{[l]} x_n^{[l]} + b_2^{[l]}$$

$$z_3^{[l]} = w_{31}^{[l]} x_1^{[l]} + w_{32}^{[l]} x_2^{[l]} + \dots + w_{3n}^{[l]} x_n^{[l]} + b_3^{[l]} \quad \text{and so on.}$$

In matrix form, it can be written as

$$\begin{pmatrix} z_1^{[l]} \\ z_2^{[l]} \\ z_3^{[l]} \\ \vdots \\ z_m^{[l]} \end{pmatrix} = \begin{pmatrix} w_{11}^{[l]} & w_{12}^{[l]} & w_{13}^{[l]} & \dots & w_{1n}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & w_{23}^{[l]} & \dots & w_{2n}^{[l]} \\ w_{31}^{[l]} & w_{32}^{[l]} & w_{33}^{[l]} & \dots & w_{3n}^{[l]} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ w_{m1}^{[l]} & w_{m2}^{[l]} & w_{m3}^{[l]} & \dots & w_{mn}^{[l]} \end{pmatrix} \times \begin{pmatrix} x_1^{[l]} \\ x_2^{[l]} \\ x_3^{[l]} \\ \vdots \\ x_n^{[l]} \end{pmatrix} + \begin{pmatrix} b_1^{[l]} \\ b_2^{[l]} \\ b_3^{[l]} \\ \vdots \\ b_n^{[l]} \end{pmatrix}$$

$$Z^{[l]} = W^{[l]}X^{[l]} + B^{[l]}$$

$$X^{[l+1]} = A(Z^{[l]})$$

3.5.3 Backpropagation Equations

Backward propagation equations are involved in gradient computations of optimization. It propagates from output layer to input layer since succeeding layer gradients are required to compute present gradient. These equations are used during training phase to optimize the loss function as discussed before. The gradient update equations in each layer for different

parameters are examined in below section. Understanding of these equations helps while designing training network.

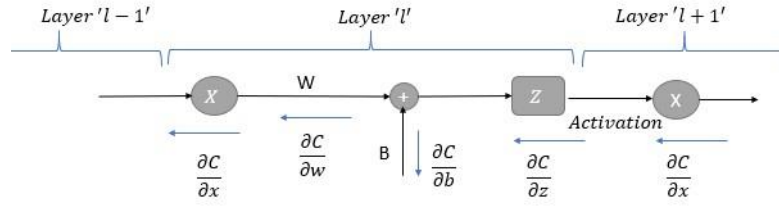


Figure 13 Backpropagation for a single neuron

For each layer ' l ',

$$\frac{\partial C}{\partial z_1^{[l]}} = \frac{\partial C}{\partial x_1^{[l+1]}} \cdot \frac{\partial x_1^{[l+1]}}{\partial z_1^{[l]}} = \frac{\partial C}{\partial x_1^{[l+1]}} \cdot A'(z_1^{[l]})$$

$$\frac{\partial C}{\partial z_2^{[l]}} = \frac{\partial C}{\partial x_2^{[l+1]}} \cdot \frac{\partial x_2^{[l+1]}}{\partial z_2^{[l]}} = \frac{\partial C}{\partial x_2^{[l+1]}} \cdot A'(z_2^{[l]})$$

This can be written as matrix equation as follows.

$$\begin{pmatrix} \frac{\partial C}{\partial z_1^{[l]}} \\ \frac{\partial C}{\partial z_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial z_m^{[l]}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial x_1^{[l+1]}} \\ \frac{\partial C}{\partial x_2^{[l+1]}} \\ \vdots \\ \frac{\partial C}{\partial x_m^{[l+1]}} \end{pmatrix} \odot \begin{pmatrix} A'(z_1^{[l]}) \\ A'(z_2^{[l]}) \\ \vdots \\ A'(z_m^{[l]}) \end{pmatrix}$$

$$\left(\frac{\partial C}{\partial Z^{[l]}} \right) = \left(\frac{\partial C}{\partial X^{[l+1]}} \right) \odot A'(Z^{[l]})$$

Where \odot operation is the elementwise product of two matrices.

3.5.4 Weight update

Using chain rule

$$\frac{\partial C}{\partial w_{11}^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial w_{11}^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot x_1$$

Similarly,

$$\frac{\partial C}{\partial w_{12}^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial w_{12}^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot x_2$$

$$\frac{\partial C}{\partial w_{21}^{[l]}} = \frac{\partial C}{\partial z_2^{[l]}} \cdot \frac{\partial z_2^{[l]}}{\partial w_{21}^{[l]}} = \frac{\partial C}{\partial z_2^{[l]}} \cdot x_1$$

$$\frac{\partial C}{\partial w_{22}^{[l]}} = \frac{\partial C}{\partial z_2^{[l]}} \cdot \frac{\partial z_2^{[l]}}{\partial w_{22}^{[l]}} = \frac{\partial C}{\partial z_2^{[l]}} \cdot x_2$$

In matrix form it can be expressed as,

$$\begin{pmatrix} \frac{\partial C}{\partial w_{11}^{[l]}} & \frac{\partial C}{\partial w_{12}^{[l]}} & \cdots & \frac{\partial C}{\partial w_{1n}^{[l]}} \\ \frac{\partial C}{\partial w_{21}^{[l]}} & \frac{\partial C}{\partial w_{22}^{[l]}} & \cdots & \frac{\partial C}{\partial w_{2n}^{[l]}} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial C}{\partial w_{m1}^{[l]}} & \frac{\partial C}{\partial w_{m2}^{[l]}} & \cdots & \frac{\partial C}{\partial w_{mn}^{[l]}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial z_1^{[l]}} \\ \frac{\partial C}{\partial z_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial z_m^{[l]}} \end{pmatrix} \times (x_1 \quad x_2 \quad \cdots \quad x_n)$$

$$\left(\frac{\partial C}{\partial w^{[l]}} \right) = \left(\frac{\partial C}{\partial z^{[l]}} \right) (X^{[l]})^T$$

Where X^T is transport of X .

3.5.5 Bias update

$$\frac{\partial C}{\partial b_1^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial b_1^{[l]}} = \frac{\partial C}{\partial z_1^{[l]}} \cdot 1 = \frac{\partial C}{\partial z_1^{[l]}}$$

Similarly,

$$\frac{\partial C}{\partial b_2^{[l]}} = \frac{\partial C}{\partial z_2^{[l]}}$$

$$\begin{pmatrix} \frac{\partial C}{\partial b_1^{[l]}} \\ \frac{\partial C}{\partial b_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial b_m^{[l]}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial z_1^{[l]}} \\ \frac{\partial C}{\partial z_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial z_m^{[l]}} \end{pmatrix}$$

i.e.

$$\left(\frac{\partial C}{\partial b^{[l]}}\right) = \left(\frac{\partial C}{\partial z^{[l]}}\right)$$

3.5.6 Previous layer update

$$\begin{aligned}\frac{\partial C}{\partial x_1^{[l]}} &= \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial x_1^{[l]}} + \frac{\partial C}{\partial z_2^{[l]}} \cdot \frac{\partial z_2^{[l]}}{\partial x_1^{[l]}} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot \frac{\partial z_m^{[l]}}{\partial x_1^{[l]}} \\ &= \frac{\partial C}{\partial z_1^{[l]}} \cdot w_{11}^{[l]} + \frac{\partial C}{\partial z_2^{[l]}} \cdot w_{21}^{[l]} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot w_{m1}^{[l]}\end{aligned}$$

Similarly,

$$\begin{aligned}\frac{\partial C}{\partial x_2^{[l]}} &= \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial x_2^{[l]}} + \frac{\partial C}{\partial z_2^{[l]}} \cdot \frac{\partial z_2^{[l]}}{\partial x_2^{[l]}} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot \frac{\partial z_m^{[l]}}{\partial x_2^{[l]}} \\ &= \frac{\partial C}{\partial z_1^{[l]}} \cdot w_{12}^{[l]} + \frac{\partial C}{\partial z_2^{[l]}} \cdot w_{22}^{[l]} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot w_{m2}^{[l]}\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial x_n^{[l]}} &= \frac{\partial C}{\partial z_1^{[l]}} \cdot \frac{\partial z_1^{[l]}}{\partial x_n^{[l]}} + \frac{\partial C}{\partial z_2^{[l]}} \cdot \frac{\partial z_2^{[l]}}{\partial x_n^{[l]}} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot \frac{\partial z_m^{[l]}}{\partial x_n^{[l]}} \\ &= \frac{\partial C}{\partial z_1^{[l]}} \cdot w_{1n}^{[l]} + \frac{\partial C}{\partial z_2^{[l]}} \cdot w_{2n}^{[l]} + \dots \dots \dots + \frac{\partial C}{\partial z_m^{[l]}} \cdot w_{mn}^{[l]}\end{aligned}$$

In matrix form it can be expressed as,

$$\begin{aligned}\begin{pmatrix} \frac{\partial C}{\partial x_1^{[l]}} \\ \frac{\partial C}{\partial x_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial x_n^{[l]}} \end{pmatrix} &= \begin{pmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \dots & \dots & w_{1n}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \dots & \dots & w_{2n}^{[l]} \\ \vdots & \vdots & \dots & \dots & \vdots \\ w_{m1}^{[l]} & w_{m2}^{[l]} & \dots & \dots & w_{mn}^{[l]} \end{pmatrix}^T \times \begin{pmatrix} \frac{\partial C}{\partial z_1^{[l]}} \\ \frac{\partial C}{\partial z_2^{[l]}} \\ \vdots \\ \frac{\partial C}{\partial z_m^{[l]}} \end{pmatrix} \\ \left(\frac{\partial C}{\partial x^{[l]}}\right) &= (W^{[l]})^T \times \left(\frac{\partial C}{\partial z^{[l]}}\right)\end{aligned}$$

These backpropagation equations in matrix form can be used in the algorithms

during training the network.

3.6 Summary

In this chapter, the fundamental concepts of machine learning were discussed. We have seen how computers can be made to learn from data to perform tasks where explicit programming is very difficult. Gradient descent optimizing algorithm and how it works in machine learning were discussed. After that commonly used activation functions in machine learning and their differences were mentioned. We have seen that improved algorithms on gradient descent can be helpful to boost convergence speed and accuracy. After that a multi-layer perceptron model and their forward and backward propagation equations for update of parameters were discussed. These equations are used to develop training and inference algorithms of the network. Basic theoretical understanding of the algorithms and approaches helps to analyse the network behaviour better and aids to improvise the performance.

4 OVERVIEW OF CONVOLUTIONAL NEURAL NETWORKS

The convolutional operation was used to extract out vital information out of a signal. It was significantly used as filters even before the machine learning algorithms came into picture. In this chapter, the basics of convolution applied to image filtering as well as incorporating convolutional layers into the deep network for the feature extraction will be discussed.

4.1 Convolution operation

It is a mathematical operation between two functions. It is used in signal processing to extract some vital information from the inputs.

Some basic conventions used in this literature are

* : Convolution operation ignoring the data points at the edges.

\odot : Complete convolution between two inputs.




\otimes : Complete cross-correlation between two inputs.

\boxtimes : Correlation ignoring the edges.

\odot : Element wise multiplication between two matrices. (Hadamard product)

Computer sees image as pixels. Pixel is the smallest element of a picture, with each pixel having an intensity value with a colour. Simplest example is that of a Gray scale image. Pixels hold information on different intensities of Gray with black as the weakest intensity and white as the strongest.

A simple convention can be as follows.

Pixel value	Image
≥ 1	
≤ -1	
0	

The output of 2D convolution (Complete convolution) between two inputs is given as

$$y[m, n] = x(m, n) \circledast h(m, n) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x(i, j) \cdot h(m - i, n - j)$$

A similar function as that of convolution is cross-correlation operation which is used to find the similarity between two inputs.

$$y[m, n] = x(m, n) \otimes h(m, n) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x(i, j) \cdot h(m + i, n + j)$$

Defining convolution and cross-correlation using above formulas may not be a good idea, because this formula assumes value of x and h outside its dimension. As '0' value for a pixel intensity could mean some colour for the pixel according to the convention used and it might lead to some improper values at the edges. So, it is better to calculate output values at those places only where there is complete overlap between the filter and the input. (Referred to as 'convolution' and 'cross-correlation').

4.1.1 Edge Detection

Consider a simple (6, 6) image I and an edge detecting filter as shown.

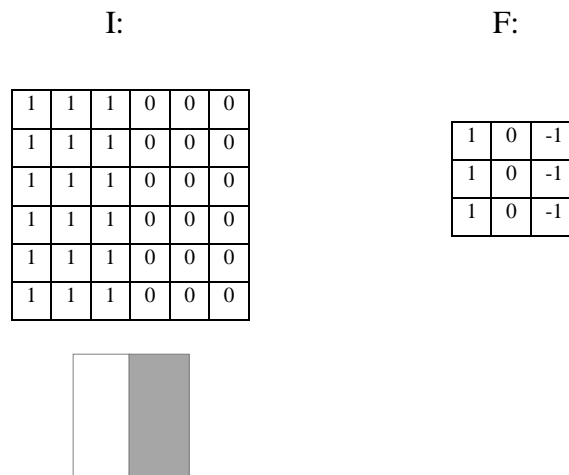


Figure 14 A sample 6x6 image and its visual representation

We can clearly see that there is an edge in the middle of the image. The values of the filter are chosen by hand specifically to determine an edge in the input.

Cross-correlating these two inputs,

$$Y = I \otimes F$$

$$\begin{aligned} y_{11} &= i_{11}f_{11} + i_{12}f_{12} + i_{13}f_{13} + i_{21}f_{21} + i_{22}f_{22} + i_{23}f_{23} \\ &\quad + i_{31}f_{31} + i_{32}f_{32} + i_{33}f_{33} \\ &= 1 + 0 + (-1) + 1 + 0 + (-1) + 1 + 0 + (-1) \\ &= 0 \end{aligned}$$

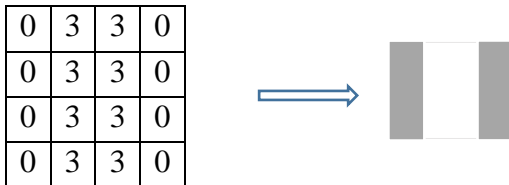
1 ¹	1 ⁰	1 ⁻¹
1 ¹	1 ⁰	1 ⁻¹
1 ¹	1 ⁰	1 ⁻¹

Similarly,

$$\begin{aligned} y_{12} &= i_{12}f_{11} + i_{13}f_{12} + i_{14}f_{13} + i_{22}f_{21} + i_{23}f_{22} \\ &\quad + i_{24}f_{23} + i_{32}f_{31} + i_{33}f_{32} + i_{34}f_{33} \\ &= 1 + 0 + 0 + 1 + 0 + 0 + 1 + 0 + 0 \\ &= 3 \end{aligned}$$

1 ¹	1 ⁰	0 ⁻¹
1 ¹	1 ⁰	0 ⁻¹
1 ¹	1 ⁰	0 ⁻¹

At every point y can be evaluated and it turns out to



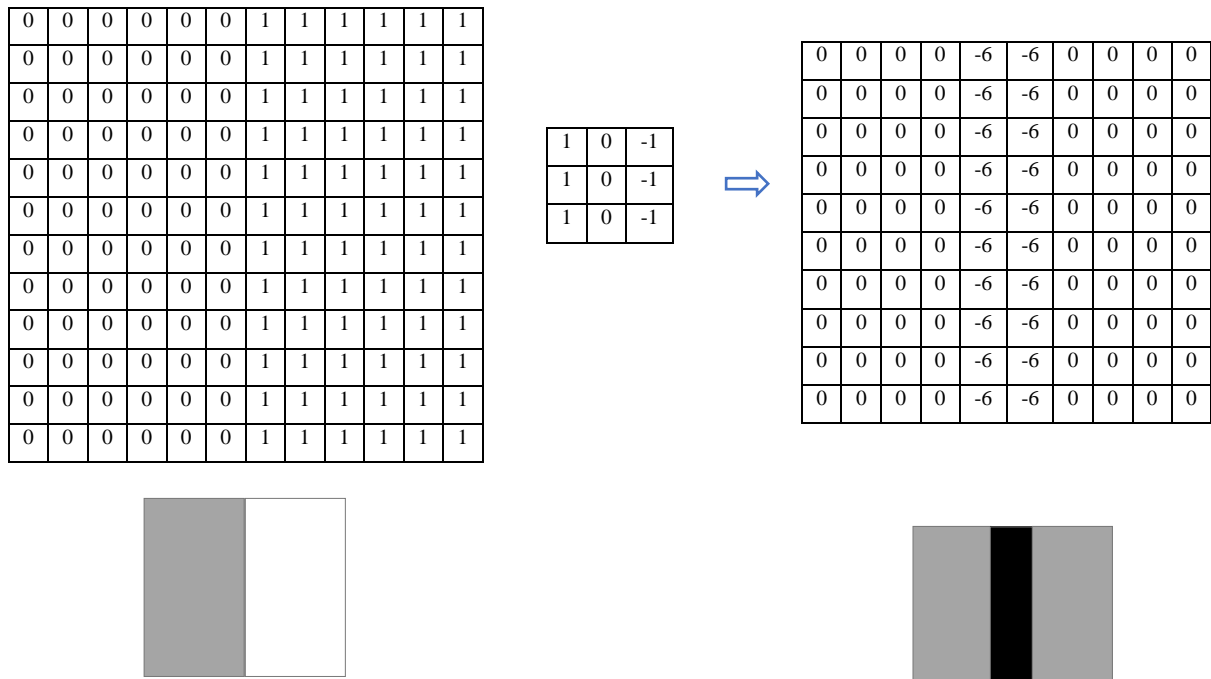
The white line in the middle indicates the edge in the original image. The grey shades on both sides of the white lines indicates that colours of similar intensity are present on the sides of the image. This result becomes more obvious as the size of the input image becomes bigger.

E.g.:

Let I_{ij} be a (12, 12) image with

$$I_{ij} = \begin{cases} 0 & \text{if } i \leq 6 \\ 1 & \text{if } i > 6 \end{cases}$$

This image is correlated with the same filter.



The thin line in the middle indicates the presence of edge at the centre of the image. The values in the filter above were handcrafted. For complex applications, handcrafting filters for extracting complex features is too difficult or next to impossible. So is the question, can computer learn these values somehow on its own?

4.2 Incorporating Convolutional layers in deep network

The computer is given training examples from which it starts to find these filter values. The filter values are initialized randomly and values of these filters are changed iteratively from the training examples in a well-informed manner. The two parameters associated with a layer in CNN are weights and biases.

MSE (Mean square error) Loss/Cost function can be used as cost function.

$$C = \frac{1}{2} \sum_j \sum_i (y_{ij}^{observed} - y_{ij}^{expected})^2$$

where, $y_{ij}^{expected}$ is the actual result which should have come as output and $y_{ij}^{observed}$ is the output from CNN. Ideally $C = 0$. But it happens only when $y_{ij}^{observed} = y_{ij}^{expected}$. The next best thing is to adjust f and b such that C is minimum.

So f and b are changed iteratively as per gradient descent method.

$$f' = f - \alpha \frac{\partial C}{\partial f}$$

$$b' = b - \alpha \frac{\partial C}{\partial b}$$

The gradient values are backpropagated through each layer which are used to update the values of f and b .

Let's see that with an example. Let's define a two-layer CNN as below.

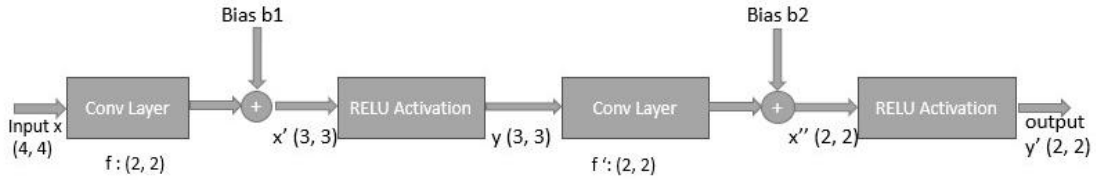


Figure 15 A sample 2- layer CNN with activation functions

Let the filters and biases are initialized as follows.

$$f = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad f' = \begin{pmatrix} -1 & 2 \\ 3 & -4 \end{pmatrix}$$

$$b_1 = 5 \quad b_2 = -10$$

Let the first training sample be as follows.

$$x_{in} = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \quad y_{expected} = \begin{pmatrix} 10 & -10 \\ 10 & -10 \end{pmatrix}$$

4.2.1 Forward Propagation

As the name suggests, the data is fed into CNN and moves forward from left to right across layers. Even though layer name is “convolutional layer”, the exact operation performed is the cross-correlation operation between input and filter.

4.2.2 Activation functions

The output from convolutional layer is passed through an activation function in CNN. For example, x' in the above neural network was passed into a RELU activation before it was passed into next convolutional layer.

$$x' = x \otimes f + b_1$$

$$x' = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{pmatrix}$$

$$= \begin{pmatrix} 10 & -2 & -10 \\ 10 & -2 & -10 \\ 10 & -2 & -10 \end{pmatrix} + \begin{pmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{pmatrix} = \begin{pmatrix} 15 & 3 & -5 \\ 15 & 3 & -5 \\ 15 & 3 & -5 \end{pmatrix}$$

$$y = Relu(x') = \begin{pmatrix} 15 & 3 & 0 \\ 15 & 3 & 0 \\ 15 & 3 & 0 \end{pmatrix}$$

$$x'' = y \otimes f' + b_2$$

$$x'' = \begin{pmatrix} 15 & 3 & 0 \\ 15 & 3 & 0 \\ 15 & 3 & 0 \end{pmatrix} \otimes \begin{pmatrix} -1 & 2 \\ 3 & -4 \end{pmatrix} + \begin{pmatrix} -10 & -10 \\ -10 & -10 \end{pmatrix} = \begin{pmatrix} 14 & -4 \\ 14 & -4 \\ 14 & -4 \end{pmatrix}$$

$$y' = Relu(x'') = \begin{pmatrix} 14 & 0 \\ 14 & 0 \end{pmatrix}$$

4.2.3 Backpropagation

Computing $\frac{\partial C}{\partial f}$ and $\frac{\partial C}{\partial b}$ terms required in update equations directly is difficult. So, gradients at the output are computed first and backpropagate using chain rule to compute these terms.

To find $\frac{\partial C}{\partial f}$ and $\frac{\partial C}{\partial b}$ terms, let's find gradient at y' is and backpropagate layer by layer.



Figure 16 Back propagating cost functions

$$C = \frac{1}{2} \sum_j \sum_i (y'_{ij} - y_{ij}^{expected})^2$$

$$\frac{\partial C}{\partial y'_{ij}} = (y'_{ij} - y_{ij}^{expected})$$

In matrix form,

$$\begin{pmatrix} \frac{\partial C}{\partial y'_{ij}} & \frac{\partial C}{\partial y'_{ij}} \\ \frac{\partial C}{\partial y'_{ij}} & \frac{\partial C}{\partial y'_{ij}} \end{pmatrix} = y' - y^{expected} = \begin{pmatrix} 14 & 0 \\ 14 & 0 \end{pmatrix} - \begin{pmatrix} 10 & -10 \\ 10 & -10 \end{pmatrix} = \begin{pmatrix} 4 & 10 \\ 4 & 10 \end{pmatrix}$$

We know that $y'_{ij} = Relu(x''_{ij})$

Therefore $\frac{\partial y'_{ij}}{\partial x''_{ij}} = Relu'(x''_{ij})$

where derivative of RELU is represented as $Relu'$.

Now $\frac{\partial C}{\partial x''_{ij}} = \frac{\partial C}{\partial y'_{ij}} \frac{\partial y'_{ij}}{\partial x''_{ij}} = \frac{\partial C}{\partial y'_{ij}} \cdot Relu'(x''_{ij})$

i.e.

$$\frac{\partial C}{\partial x''_{11}} = \frac{\partial C}{\partial y'_{11}} \frac{\partial y'_{11}}{\partial x''_{11}} = \frac{\partial C}{\partial y'_{11}} \cdot Relu'(x''_{11})$$

$$\frac{\partial C}{\partial x''_{12}} = \frac{\partial C}{\partial y'_{12}} \frac{\partial y'_{12}}{\partial x''_{12}} = \frac{\partial C}{\partial y'_{12}} \cdot Relu'(x''_{12})$$

$$\frac{\partial C}{\partial x''_{21}} = \frac{\partial C}{\partial y'_{21}} \frac{\partial y'_{21}}{\partial x''_{21}} = \frac{\partial C}{\partial y'_{21}} \cdot Relu'(x''_{21})$$

$$\frac{\partial C}{\partial x''_{22}} = \frac{\partial C}{\partial y'_{22}} \frac{\partial y'_{22}}{\partial x''_{22}} = \frac{\partial C}{\partial y'_{22}} \cdot Relu'(x''_{22})$$

it can be represented as a matrix equation as

$$\frac{\partial C}{\partial x''} = \frac{\partial C}{\partial y'} \odot Relu'(x'')$$

In general,

$$\frac{\partial C}{\partial x''} = \frac{\partial C}{\partial y'} \odot Activation'(x'')$$

Where \odot denotes element wise product of two matrices and $Activation'$ denotes derivative of activation function.

$$Relu'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Therefore,

$$\frac{\partial C}{\partial x''} = \begin{pmatrix} 4 & 10 \\ 4 & 10 \end{pmatrix} \odot \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 4 & 0 \\ 4 & 0 \end{pmatrix}$$

We know that

$$x'' = y \otimes f' + b_2$$

i.e.

$$x''_{11} = y_{11}f'_{11} + y_{12}f'_{12} + y_{21}f'_{21} + y_{22}f'_{22} + b_2$$

$$x''_{12} = y_{12}f'_{11} + y_{13}f'_{12} + y_{22}f'_{21} + y_{23}f'_{22} + b_2$$

$$x''_{21} = y_{21}f'_{11} + y_{22}f'_{12} + y_{31}f'_{21} + y_{32}f'_{22} + b_2$$

$$x''_{22} = y_{22}f'_{11} + y_{23}f'_{12} + y_{32}f'_{21} + y_{33}f'_{22} + b_2$$

Evaluating $\frac{\partial C}{\partial b_2}$,

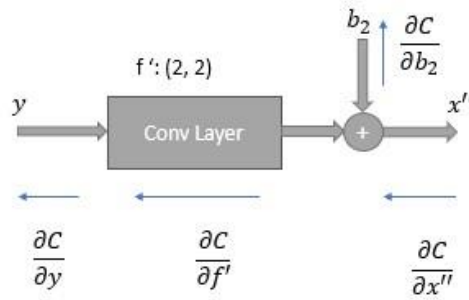


Figure 17 Back propagating bias and its updating

Applying chain rule

$$\frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial x''_{11}} \cdot \frac{\partial x''_{11}}{\partial b_2} + \frac{\partial C}{\partial x''_{12}} \cdot \frac{\partial x''_{12}}{\partial b_2} + \frac{\partial C}{\partial x''_{21}} \cdot \frac{\partial x''_{21}}{\partial b_2} + \frac{\partial C}{\partial x''_{22}} \cdot \frac{\partial x''_{22}}{\partial b_2}$$

Since $\frac{\partial x''_{ij}}{\partial b_2} = 1$

$$\frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial x''_{11}} + \frac{\partial C}{\partial x''_{12}} + \frac{\partial C}{\partial x''_{21}} + \frac{\partial C}{\partial x''_{22}}$$

In general,

$$\frac{\partial C}{\partial b} = \sum_j \sum_i \frac{\partial C}{\partial x''_{ij}}$$

In the example,

$$\frac{\partial C}{\partial b_2} = 4 + 4 + 0 + 0 = 8$$

Updated $b_2 = b_2 - \alpha \frac{\partial C}{\partial b_2}$ (Let $\alpha = 1$)

$$= -10 - 8 = -18$$

This will be the value of b_2 after first iteration.

4.2.4 Filter values update

Applying chain rule,

$$\frac{\partial C}{\partial f'_{11}} = \frac{\partial C}{\partial x''_{11}} \cdot \frac{\partial x''_{11}}{\partial f'_{11}} + \frac{\partial C}{\partial x''_{12}} \cdot \frac{\partial x''_{12}}{\partial f'_{11}} + \frac{\partial C}{\partial x''_{21}} \cdot \frac{\partial x''_{21}}{\partial f'_{11}} + \frac{\partial C}{\partial x''_{22}} \cdot \frac{\partial x''_{22}}{\partial f'_{11}}$$

Implies,

$$\frac{\partial C}{\partial f'_{11}} = \frac{\partial C}{\partial x''_{11}} y_{11} + \frac{\partial C}{\partial x''_{12}} y_{12} + \frac{\partial C}{\partial x''_{21}} y_{21} + \frac{\partial C}{\partial x''_{22}} y_{22}$$

Similarly,

$$\frac{\partial C}{\partial f'_{12}} = \frac{\partial C}{\partial x''_{11}} y_{12} + \frac{\partial C}{\partial x''_{12}} y_{13} + \frac{\partial C}{\partial x''_{21}} y_{22} + \frac{\partial C}{\partial x''_{22}} y_{23}$$

$$\frac{\partial C}{\partial f'_{21}} = \frac{\partial C}{\partial x''_{11}} y_{21} + \frac{\partial C}{\partial x''_{12}} y_{22} + \frac{\partial C}{\partial x''_{21}} y_{31} + \frac{\partial C}{\partial x''_{22}} y_{32}$$

$$\frac{\partial C}{\partial f'_{22}} = \frac{\partial C}{\partial x''_{11}} y_{22} + \frac{\partial C}{\partial x''_{12}} y_{23} + \frac{\partial C}{\partial x''_{21}} y_{32} + \frac{\partial C}{\partial x''_{22}} y_{33}$$

This can be written in matrix form as,

$$\begin{pmatrix} \frac{\partial C}{\partial f'_{11}} & \frac{\partial C}{\partial f'_{12}} \\ \frac{\partial C}{\partial f'_{21}} & \frac{\partial C}{\partial f'_{22}} \end{pmatrix} = \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix} \otimes \begin{pmatrix} \frac{\partial C}{\partial x''_{11}} & \frac{\partial C}{\partial x''_{12}} \\ \frac{\partial C}{\partial x''_{21}} & \frac{\partial C}{\partial x''_{22}} \end{pmatrix}$$

In general, $\left(\frac{\partial C}{\partial f'}\right) = y \otimes \left(\frac{\partial C}{\partial x''}\right)$

Where \otimes is cross-correlation.

In the example,

$$\begin{pmatrix} \frac{\partial C}{\partial f'_{11}} & \frac{\partial C}{\partial f'_{12}} \\ \frac{\partial C}{\partial f'_{21}} & \frac{\partial C}{\partial f'_{22}} \end{pmatrix} = \begin{pmatrix} 15 & 3 & 0 \\ 15 & 3 & 0 \\ 15 & 3 & 0 \end{pmatrix} \otimes \begin{pmatrix} 4 & 0 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 120 & 24 \\ 120 & 24 \end{pmatrix}$$

Filter values will be updated as

$$\text{Updated } f' = f' - \alpha \frac{\partial C}{\partial f'} \quad (\text{let } \alpha = 1)$$

$$= \begin{pmatrix} -1 & 2 \\ 3 & -4 \end{pmatrix} - \begin{pmatrix} 120 & 24 \\ 120 & 24 \end{pmatrix} = \begin{pmatrix} -121 & -22 \\ -117 & -28 \end{pmatrix}$$

4.2.5 Gradient to Previous Layer Evaluation

$$\frac{\partial C}{\partial y_{11}} = \frac{\partial C}{\partial x''_{11}} \cdot \frac{\partial x''_{11}}{\partial y_{11}} = \frac{\partial C}{\partial x''_{11}} f'_{11}$$

$$\frac{\partial C}{\partial y_{12}} = \frac{\partial C}{\partial x''_{11}} \cdot \frac{\partial x''_{11}}{\partial y_{12}} + \frac{\partial C}{\partial x''_{12}} \cdot \frac{\partial x''_{12}}{\partial y_{12}} = \frac{\partial C}{\partial x''_{11}} f'_{12} + \frac{\partial C}{\partial x''_{12}} f'_{11}$$

Simplifying remaining terms,

$$\frac{\partial C}{\partial y_{13}} = \frac{\partial C}{\partial x''_{12}} f'_{12}$$

$$\frac{\partial C}{\partial y_{21}} = \frac{\partial C}{\partial x''_{11}} f'_{21} + \frac{\partial C}{\partial x''_{21}} f'_{11}$$

$$\frac{\partial C}{\partial y_{22}} = \frac{\partial C}{\partial x''_{11}} f'_{22} + \frac{\partial C}{\partial x''_{12}} f'_{21} + \frac{\partial C}{\partial x''_{21}} f'_{12} + \frac{\partial C}{\partial x''_{22}} f'_{11}$$

$$\frac{\partial C}{\partial y_{23}} = \frac{\partial C}{\partial x''_{12}} f'_{22} + \frac{\partial C}{\partial x''_{22}} f'_{12}$$

$$\frac{\partial C}{\partial y_{31}} = \frac{\partial C}{\partial x''_{21}} f'_{21}$$

$$\frac{\partial C}{\partial y_{32}} = \frac{\partial C}{\partial x''_{21}} f'_{22} + \frac{\partial C}{\partial x''_{22}} f'_{21}$$

$$\frac{\partial C}{\partial y_{33}} = \frac{\partial C}{\partial x''_{22}} f'_{22}$$

This can be simplified as,

$$\begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} & \frac{\partial C}{\partial y_{13}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} & \frac{\partial C}{\partial y_{23}} \\ \frac{\partial C}{\partial y_{31}} & \frac{\partial C}{\partial y_{32}} & \frac{\partial C}{\partial y_{33}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial x''_{11}} & \frac{\partial C}{\partial x''_{12}} \\ \frac{\partial C}{\partial x''_{21}} & \frac{\partial C}{\partial x''_{22}} \end{pmatrix} \odot \begin{pmatrix} f'_{11} & f'_{12} \\ f'_{21} & f'_{22} \end{pmatrix}$$

In general, $\left(\frac{\partial C}{\partial y}\right) = \left(\frac{\partial C}{\partial x}\right) \odot f'$

Where \odot is complete convolution.

$$\text{In the example } \left(\frac{\partial C}{\partial y}\right) = \begin{pmatrix} 4 & 0 \\ 4 & 0 \end{pmatrix} \odot \begin{pmatrix} -1 & 2 \\ 3 & -4 \end{pmatrix} = \begin{pmatrix} -4 & 8 & 0 \\ 8 & -8 & 0 \\ 12 & -16 & 0 \end{pmatrix}$$

Back propagating to previous layer in the example can be computed as follows.

Figure

$$\begin{aligned} \left(\frac{\partial C}{\partial x'}\right) &= \left(\frac{\partial C}{\partial y}\right) \odot Relu'(x') \\ &= \begin{pmatrix} -4 & 8 & 0 \\ 8 & -8 & 0 \\ 12 & -16 & 0 \end{pmatrix} \odot \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} -4 & 8 & 0 \\ 8 & -8 & 0 \\ 12 & -16 & 0 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \frac{\partial C}{\partial b_1} &= \sum_j \sum_i \frac{\partial C}{\partial x'_{ij}} = -4 + 8 + 8 + (-8) + 12 + (-16) \\ &= -8 \end{aligned}$$

$$b_1 = b_1 - \alpha \frac{\partial C}{\partial b_1} = 5 - 1 \times (-8) = 13$$

$$\left(\frac{\partial C}{\partial f}\right) = x \otimes \left(\frac{\partial C}{\partial x'}\right) = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \otimes \begin{pmatrix} -4 & 8 & 0 \\ 8 & -8 & 0 \\ 12 & -16 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 32 \\ 0 & 32 \end{pmatrix}$$

$$f = f - \alpha \frac{\partial C}{\partial f} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} 0 & 32 \\ 0 & 32 \end{pmatrix} = \begin{pmatrix} 1 & -30 \\ 3 & -28 \end{pmatrix}$$

These are the updated values of f after first iteration. The network is trained with many such training data which constantly improve the values of the filter.

4.3 Pooling Layers

Pooling layers are used in CNN to down sample feature map and thereby reduce the total number of computations required in forward and backward propagation.

Consider an architecture with only Convolutional and fully connected layers.

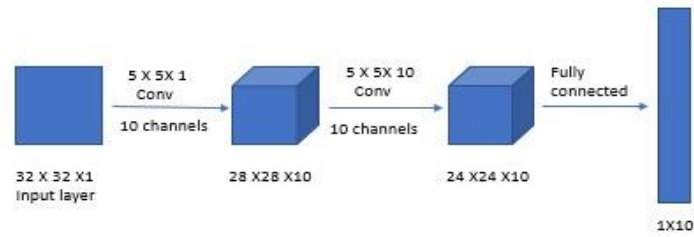


Figure 18 A sample 2-layer CNN to demonstrate number of computations

Here total number of computations (multiplications) in forward propagation

$$\begin{aligned}
 &= (5 \times 5 \times 1) \times (28 \times 28 \times 10) + (5 \times 5 \times 10) \times \\
 &(24 \times 24 \times 10) + (24 \times 24 \times 10) \times 10 = 1693600 \\
 &\approx 1.7 M
 \end{aligned}$$

Two of the commonly used pooling methods are

4.3.1 Max Pooling

(f, f) window is slid over the input and maximum value is stored as the output.

$$\text{E.g.: } \begin{pmatrix} 2 & 3 & 1 & 9 \\ 4 & 7 & 3 & 5 \\ 8 & 2 & 2 & 2 \\ 1 & 3 & 4 & 5 \end{pmatrix} \xrightarrow{\text{Max pooling } f=2} \begin{pmatrix} 7 & 9 \\ 8 & 5 \end{pmatrix}$$

$$7 = \max\{2, 3, 4, 7\}$$

4.3.2 Average Pooling

(f, f) window is slid over the input and average value is stored as the output.

$$\text{E.g.: } \begin{pmatrix} 2 & 3 & 1 & 9 \\ 4 & 7 & 3 & 5 \\ 8 & 2 & 2 & 2 \\ 1 & 3 & 4 & 5 \end{pmatrix} \xrightarrow{\text{Average pooling } f=2} \begin{pmatrix} 4 & 4.5 \\ 3.25 & 3.25 \end{pmatrix}$$

$$4 = \frac{1}{4}(2 + 3 + 4 + 7)$$

To see the effect of pooling layers in reducing number of computations, Consider the architecture in previous example modified by adding pooling layers after each convolutional layer.

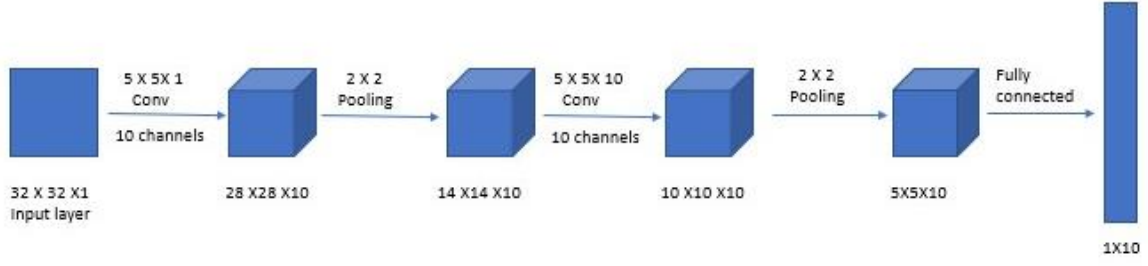


Figure 19 A sample 2-layer CNN to demonstrate number of computations after pooling

$$\begin{aligned} \text{Now total number of computations (multiplications)} &= (5 \times 5 \times 1) \times (28 \times 28 \times 10) + \\ &\quad (5 \times 5 \times 10) \times ((10 \times 10 \times 10) + 10) = 272100 \\ &\approx 2.7 L \end{aligned}$$

Number of computations are reduced significantly here. So, by adding pooling layers once in a while after convolutional layers reduces the dimension and hence the computations required. The type of pooling suitable for CNN might depend upon the application where it is used and the nature of the dataset chosen. For example, in MNIST dataset for digit recognition, digits are represented with high pixel values (white) and background is low pixel values (black), so usage of max pooling can be justified logically as the necessary information is retained after sampling because maximum value pixels are the ones decisive in determining output.

Effect of pooling layers on back propagation is examined below. Consider an input image of size

(6, 6) and a filter of size (2, 2) performed cross-correlation and the result is passed through (2, 2) pooling layer as shown below.

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} \end{pmatrix} \otimes \begin{pmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{pmatrix}$$

$$\xrightarrow{\text{conv}} \begin{pmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{pmatrix} \xrightarrow{\text{actvtn } A} \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \xrightarrow{2 \times 2 \text{ pooling}} \begin{pmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{pmatrix}$$

For average pooling,

$$z_{11} = \frac{1}{4}(y_{11} + y_{12} + y_{21} + y_{22})$$

$$\frac{\partial C}{\partial y_{11}} = \frac{\partial C}{\partial z_{11}} \cdot \frac{\partial z_{11}}{\partial y_{11}} \quad \frac{\partial z_{11}}{\partial y_{11}} = \frac{\partial z_{11}}{\partial y_{12}} = \frac{\partial z_{11}}{\partial y_{21}} = \frac{\partial z_{11}}{\partial y_{22}} = \frac{1}{4}$$

$$\text{Therefore } \frac{\partial C}{\partial y_{11}} = \frac{1}{4} \frac{\partial C}{\partial z_{11}}$$

Similarly,

$$\frac{\partial C}{\partial y_{12}} = \frac{1}{4} \frac{\partial C}{\partial z_{11}}, \quad \frac{\partial C}{\partial y_{13}} = \frac{1}{4} \frac{\partial C}{\partial z_{11}}, \quad \frac{\partial C}{\partial y_{14}} = \frac{1}{4} \frac{\partial C}{\partial z_{11}}$$

In matrix form it can be written as

$$\begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} \end{pmatrix} = \begin{pmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{pmatrix} \cdot \frac{\partial C}{\partial z_{11}}$$

Earlier for conv layers the following equation was obtained for convolution layers.

$$\begin{pmatrix} \frac{\partial C}{\partial f_{11}} & \frac{\partial C}{\partial f_{12}} \\ \frac{\partial C}{\partial f_{21}} & \frac{\partial C}{\partial f_{22}} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} \end{pmatrix} \otimes A'(G) \cdot \begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} & \frac{\partial C}{\partial y_{13}} & \frac{\partial C}{\partial y_{14}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} & \frac{\partial C}{\partial y_{23}} & \frac{\partial C}{\partial y_{24}} \\ \frac{\partial C}{\partial y_{31}} & \frac{\partial C}{\partial y_{32}} & \frac{\partial C}{\partial y_{33}} & \frac{\partial C}{\partial y_{34}} \\ \frac{\partial C}{\partial y_{41}} & \frac{\partial C}{\partial y_{42}} & \frac{\partial C}{\partial y_{43}} & \frac{\partial C}{\partial y_{44}} \end{pmatrix}$$

Where ' \otimes ' denotes convolution operation, ' \cdot ' Denotes elementwise product and $A'(G)$ is the derivative of activation for each element in the G matrix.

Now,

$$\begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} & \frac{\partial C}{\partial y_{13}} & \frac{\partial C}{\partial y_{14}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} & \frac{\partial C}{\partial y_{23}} & \frac{\partial C}{\partial y_{24}} \\ \frac{\partial C}{\partial y_{31}} & \frac{\partial C}{\partial y_{32}} & \frac{\partial C}{\partial y_{33}} & \frac{\partial C}{\partial y_{34}} \\ \frac{\partial C}{\partial y_{41}} & \frac{\partial C}{\partial y_{42}} & \frac{\partial C}{\partial y_{43}} & \frac{\partial C}{\partial y_{44}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial z_{11}} & \frac{\partial C}{\partial z_{12}} \\ \frac{\partial C}{\partial z_{21}} & \frac{\partial C}{\partial z_{22}} \end{pmatrix} \star \begin{pmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{pmatrix}$$

Here \star operation is defined as follows. each element of first matrix is multiplied by corresponding submatrix D_{ij} of second matrix to generate a bigger sized matrix.

$$D_{ij} = \begin{pmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{pmatrix} \text{ for average pooling of } (2, 2).$$

Since $\frac{\partial C}{\partial y_{ij}}$ is obtained in backpropagation, $\frac{\partial C}{\partial f}$, $\frac{\partial C}{\partial b}$ and $\frac{\partial C}{\partial x}$ can be calculated as per the equation in convolution backpropagation.

If max pooling is used instead of average pooling, D_{ij} can be represented in the form $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ if (2, 1) indexed element in the pooling window is of maximum value. Because in backpropagation, maximum value node contributes to the derivative.

For example,

$$\begin{pmatrix} 2 & 3 & 1 & 9 \\ 4 & 7 & 3 & 5 \\ 8 & 2 & 2 & 2 \\ 1 & 3 & 4 & 5 \end{pmatrix} \xrightarrow{f=2 \text{ maxpooling}} \begin{pmatrix} 7 & 9 \\ 8 & 5 \end{pmatrix}$$

Here,

$$\begin{pmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{pmatrix} \text{ will have } D_{11} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, D_{12} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix},$$

$$D_{21} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, D_{22} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

Rest of the operation is same as that of average pooling. Therefore, backpropagation in average pooling and max pooling can be pictorially represented as below.

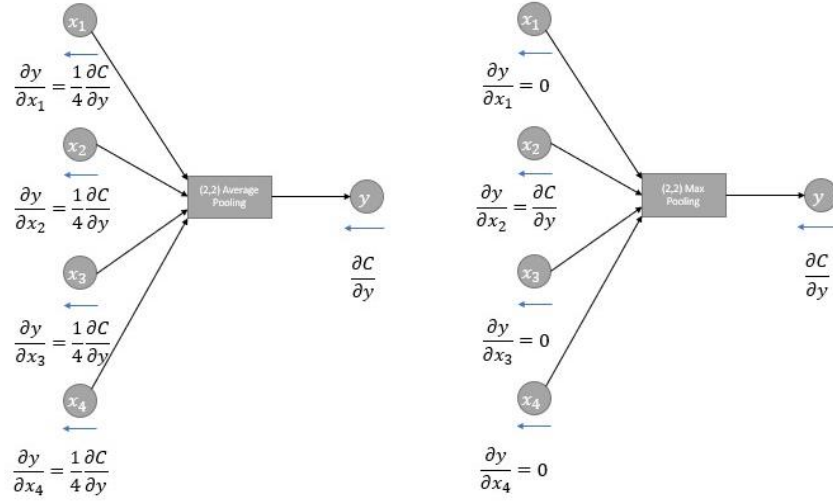


Figure 20 Backpropagation with average pooling and maxpooling

Where x_1, x_2, x_3, x_4 denotes input nodes to (2, 2) pooling and y denotes Output node with x_2 node assumed to have maximum value.

Unlike convolutional layers pooling layers do not have any parameter of its own to optimize.

4.4 Strides in Convolution

It is also used to reduce dimension of deeper layers and hence the number of computations. Filter window is stridden after a fixed gap instead of immediate row/column. Consider modified form of previous architecture where stride of more than 1 ($s=2$) is used in convolutional layers.

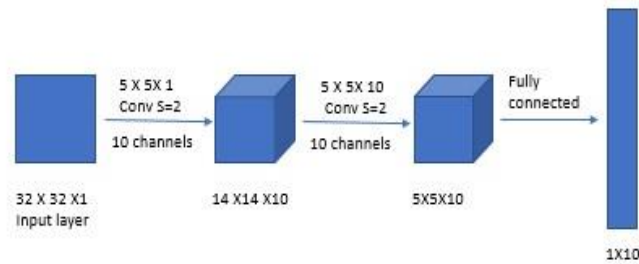


Figure 21 A sample 2-layer CNN

Total number of computations (multiplications) = $(14 \times 14 \times 10) \times (5 \times 5 \times 1)$

$$+ (5 \times 5 \times 10) \times ((5 \times 5 \times 10) + 10) = 81500$$

$$\approx 81K$$

It is evident from this example that strides in convolution saves the number of computations.

If input dimension is very large, using stride of high value is justified. For example, if the image size is 1024 X 1024, then extraction of features wouldn't get affected so much even if stride of S=2 or S=3 is used.

An example of a simple cross-correlation of 5 X 5 input with 3 X 3 filter with stride S=2 is shown below.

$$\begin{pmatrix} 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 9 & -45 \\ 9 & -45 \end{pmatrix}$$

4.4.1 Backpropagation with stride

Consider a cross-correlation of (6, 6) input with (3, 3) filter with stride s = 2.

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} \end{pmatrix} \otimes \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix}$$

$$y_{11} = f_{11}x_{11} + f_{12}x_{12} + f_{13}x_{13} + f_{21}x_{21} + f_{22}x_{22} + f_{23}x_{23} + f_{31}x_{31} + f_{32}x_{32} + f_{33}x_{33} + b$$

$$y_{12} = f_{11}x_{13} + f_{12}x_{14} + f_{13}x_{15} + f_{21}x_{23} + f_{22}x_{24} + f_{23}x_{25} + f_{31}x_{33} + f_{32}x_{34} + f_{33}x_{35} + b$$

$$y_{21} = f_{11}x_{31} + f_{12}x_{32} + f_{13}x_{33} + f_{21}x_{41} + f_{22}x_{42} + f_{23}x_{43} + f_{31}x_{51} + f_{32}x_{52} + f_{33}x_{53} + b$$

$$y_{22} = f_{11}x_{33} + f_{12}x_{34} + f_{13}x_{35} + f_{21}x_{43} + f_{22}x_{44} + f_{23}x_{45} + f_{31}x_{53} + f_{32}x_{54} + f_{33}x_{55} + b$$

$$\begin{aligned} \frac{\partial C}{\partial f_{11}} &= \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial f_{11}} + \frac{\partial C}{\partial y_{12}} \frac{\partial y_{12}}{\partial f_{11}} + \frac{\partial C}{\partial y_{21}} \frac{\partial y_{21}}{\partial f_{11}} + \frac{\partial C}{\partial y_{22}} \frac{\partial y_{22}}{\partial f_{11}} \\ &= \frac{\partial C}{\partial y_{11}} x_{11} + \frac{\partial C}{\partial y_{12}} x_{13} + \frac{\partial C}{\partial y_{21}} x_{31} + \frac{\partial C}{\partial y_{22}} x_{33} \end{aligned}$$

In matrix form this will be represented as

$$\frac{\partial C}{\partial f_{13}} = \begin{pmatrix} x_{11} & x_{13} \\ x_{31} & x_{33} \end{pmatrix} \cdot \begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} \end{pmatrix}$$

Similarly,

$$\begin{aligned} \frac{\partial C}{\partial f_{12}} &= \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial f_{12}} + \frac{\partial C}{\partial y_{12}} \frac{\partial y_{12}}{\partial f_{12}} + \frac{\partial C}{\partial y_{21}} \frac{\partial y_{21}}{\partial f_{12}} + \frac{\partial C}{\partial y_{22}} \frac{\partial y_{22}}{\partial f_{12}} \\ &= \frac{\partial C}{\partial y_{11}} x_{12} + \frac{\partial C}{\partial y_{12}} x_{14} + \frac{\partial C}{\partial y_{21}} x_{32} + \frac{\partial C}{\partial y_{22}} x_{34} \end{aligned}$$

To represent $\frac{\partial C}{\partial f}$ in matrix operation, rows and columns of zeros are inserted to $\frac{\partial C}{\partial y}$ matrix wherever convolution is skipped in stride.

i.e.

$$\begin{pmatrix} \frac{\partial C}{\partial y_{11}} & \frac{\partial C}{\partial y_{12}} \\ \frac{\partial C}{\partial y_{21}} & \frac{\partial C}{\partial y_{22}} \end{pmatrix} \xrightarrow{\text{modified to}} \begin{pmatrix} \frac{\partial C}{\partial y_{11}} & 0 & \frac{\partial C}{\partial y_{12}} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{\partial C}{\partial y_{21}} & 0 & \frac{\partial C}{\partial y_{22}} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Here zeros are added in second and fourth row and second and fourth column because while performing cross-correlation with stride s=2, second and fourth convolution window column wise and row wise were skipped.

So, update equations will be

$$\begin{pmatrix} \frac{\partial C}{\partial f_{11}} & \frac{\partial C}{\partial f_{12}} & \frac{\partial C}{\partial f_{13}} \\ \frac{\partial C}{\partial f_{21}} & \frac{\partial C}{\partial f_{22}} & \frac{\partial C}{\partial f_{23}} \\ \frac{\partial C}{\partial f_{31}} & \frac{\partial C}{\partial f_{32}} & \frac{\partial C}{\partial f_{33}} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} & x_{16} \\ x_{21} & x_{22} & x_{23} & x_{24} & x_{25} & x_{26} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} & x_{36} \\ x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} \\ x_{51} & x_{52} & x_{53} & x_{54} & x_{55} & x_{56} \\ x_{61} & x_{62} & x_{63} & x_{64} & x_{65} & x_{66} \end{pmatrix} \otimes \begin{pmatrix} \frac{\partial C}{\partial y_{11}} & 0 & \frac{\partial C}{\partial y_{12}} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{\partial C}{\partial y_{21}} & 0 & \frac{\partial C}{\partial y_{22}} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Where \otimes is cross-correlation operation.

$\frac{\partial C}{\partial b}$ equation will be sum of all $\frac{\partial C}{\partial y_{ij}}$.

4.4.2 Previous layer update

This will can also be written with modified $\frac{\partial C}{\partial y_{ij}}$ matrix with zeros inserted.

$$\begin{pmatrix} \frac{\partial C}{\partial x_{11}} & \frac{\partial C}{\partial x_{12}} & \frac{\partial C}{\partial x_{13}} & \frac{\partial C}{\partial x_{14}} & \frac{\partial C}{\partial x_{15}} & \frac{\partial C}{\partial x_{16}} \\ \frac{\partial C}{\partial x_{21}} & \frac{\partial C}{\partial x_{22}} & \frac{\partial C}{\partial x_{23}} & \frac{\partial C}{\partial x_{24}} & \frac{\partial C}{\partial x_{25}} & \frac{\partial C}{\partial x_{26}} \\ \frac{\partial C}{\partial x_{31}} & \frac{\partial C}{\partial x_{32}} & \frac{\partial C}{\partial x_{33}} & \frac{\partial C}{\partial x_{34}} & \frac{\partial C}{\partial x_{35}} & \frac{\partial C}{\partial x_{36}} \\ \frac{\partial C}{\partial x_{41}} & \frac{\partial C}{\partial x_{42}} & \frac{\partial C}{\partial x_{43}} & \frac{\partial C}{\partial x_{44}} & \frac{\partial C}{\partial x_{45}} & \frac{\partial C}{\partial x_{46}} \\ \frac{\partial C}{\partial x_{51}} & \frac{\partial C}{\partial x_{52}} & \frac{\partial C}{\partial x_{53}} & \frac{\partial C}{\partial x_{54}} & \frac{\partial C}{\partial x_{55}} & \frac{\partial C}{\partial x_{56}} \\ \frac{\partial C}{\partial x_{61}} & \frac{\partial C}{\partial x_{62}} & \frac{\partial C}{\partial x_{63}} & \frac{\partial C}{\partial x_{64}} & \frac{\partial C}{\partial x_{65}} & \frac{\partial C}{\partial x_{66}} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial y_{11}} & 0 & \frac{\partial C}{\partial y_{12}} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{\partial C}{\partial y_{21}} & 0 & \frac{\partial C}{\partial y_{22}} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \odot \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix}$$

Where \odot performs complete convolution.

It can be verified by checking some elements from this output matrix.

From equation of $y_{11}, y_{12}, y_{21}, y_{22}$ obtained earlier,

$$\frac{\partial C}{\partial x_{11}} = \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{11}} = \frac{\partial C}{\partial y_{11}} f_{11}$$

$$\frac{\partial C}{\partial x_{12}} = \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{12}} = \frac{\partial C}{\partial y_{11}} f_{12}$$

$$\frac{\partial C}{\partial x_{13}} = \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{13}} + \frac{\partial C}{\partial y_{12}} \frac{\partial y_{12}}{\partial x_{13}} = \frac{\partial C}{\partial y_{11}} f_{13} + \frac{\partial C}{\partial y_{12}} f_{11}$$

$$\frac{\partial C}{\partial x_{21}} = \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{21}} = \frac{\partial C}{\partial y_{11}} f_{21}$$

$$\begin{aligned} \frac{\partial C}{\partial x_{33}} &= \frac{\partial C}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{33}} + \frac{\partial C}{\partial y_{12}} \frac{\partial y_{12}}{\partial x_{33}} + \frac{\partial C}{\partial y_{21}} \frac{\partial y_{21}}{\partial x_{33}} + \frac{\partial C}{\partial y_{22}} \frac{\partial y_{22}}{\partial x_{33}} \\ &= \frac{\partial C}{\partial y_{11}} f_{33} + \frac{\partial C}{\partial y_{12}} f_{31} + \frac{\partial C}{\partial y_{21}} f_{13} + \frac{\partial C}{\partial y_{22}} f_{11} \end{aligned}$$

Therefore, Backpropagation with stride $s = 2$ can pictorially be represented as

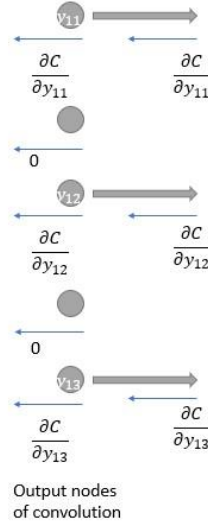


Figure 22 Backpropagation with stride

4.5 Padding in convolution

When convolution (cross-correlation to be exact) is performed, pixels deep inside are used a greater number of times than those at the edges. Hence information at edges is less represented in convolution output. To overcome this issue and also to make output size same as that of input image size, extra rows and columns are added around edges before convolution is performed.

Type of padding has to be chosen suitably in accordance with the application. For example, in MNIST dataset for handwritten digit classification, numbers are in white (high pixel value) and background is in black (0-pixel value). So, in this case suitable value to pad with is zeroes as it is the background value.

Consider an example of cross-correlation between (5, 5) input and (3, 3) filter without any padding.

$$\begin{pmatrix} 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 9 & -21 & -45 \\ 9 & -21 & -45 \\ 9 & -21 & -45 \end{pmatrix}$$

Input image size is (5, 5) but output size is reduced to (3, 3). To compare feature location and make output size same as input rows and columns of zeros are padded as shown below.

p=1

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & -1 & -1 & -1 & 0 \\ 0 & 1 & 1 & -1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 28 & 9 & -17 & -39 & -24 \\ 33 & 9 & -21 & -45 & -27 \\ 33 & 9 & -21 & -45 & -27 \\ 33 & 9 & -21 & -45 & -27 \\ 33 & 9 & -21 & -45 & -27 \\ 16 & 3 & -11 & -21 & -12 \end{pmatrix}$$

Padding is commonly used when filter size is large, say (7, 7). Because reduction in output size and loss of information at edges will be more in that scenario. It can also be used when crucial information is present at the edges to extract it out. In backpropagation with padding, $\frac{\partial C}{\partial x}$ matrix obtained will be of higher size than X matrix. the rows and columns in $\frac{\partial C}{\partial x}$ matrix correspond to padding are ignored and remaining elements of matrix are used for backpropagating to previous layer. For example, in convolution of (5, 5) input with (3, 3) filter with padding $p = 1$, $\frac{\partial C}{\partial x}$ matrix obtained by complete convolution will be of size (7, 7) but (5, 5) sized matrix after ignoring first and last row and column will be backpropagated for previous layer update.

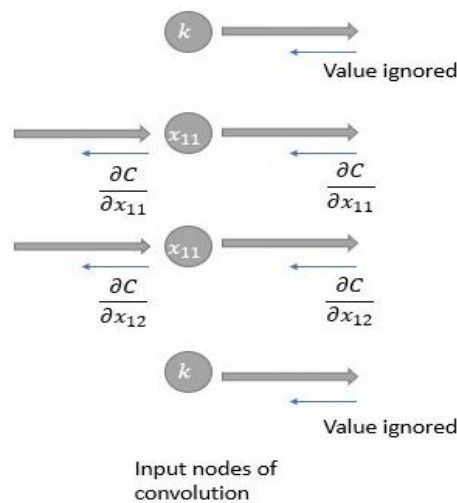


Figure 23 Backpropagation with padding

If cross-correlation is performed between (n, n) input and (f, f) filter with stride 's' and padding 'p',

Then dimension of output = $\left[\frac{n-f+1+2p}{s} \right] \times \left[\frac{n-f+1+2p}{s} \right]$

Where $\lceil x \rceil$ performs ceiling function on x (least Integer which is greater than x).

For example, if cross-correlation is performed between (6, 6) input and (2, 2) filter with stride s=2 and padding p=1,

Then output dimension = $\left\lceil \frac{6-2+1+2}{2} \right\rceil \times \left\lceil \frac{6-2+1+2}{2} \right\rceil = (4 \times 4)$.

4.6 Summary

In this chapter deep learning networks using convolutions for applications involving images were discussed. We have seen how filters using convolutions were able to extract underlying features of an image and how these filter values can be parametrized for computers to learn on its own. Updating filter weights and biases iteratively in a convolutional network is examined with their forward and backward propagation equations and an illustrative example. After that, we have discussed about Pooling layers which are used in CNN and why it is needed. We have seen the roles of Strides and padding in convolutions, and how they help to reduce computations and avoid loss of informations at edges of image. The modification in backward propagation equations with strides and padding was also examined. These basic intuitive understanding is helpful while using convolutional neural networks for the desired application and tuning the network to improve the performance.

5 CHARACTER RECOGNITION USING NEURAL NETWORKS

In this chapter, different architectures are implemented for character recognition. The parameters to measure performance will be defined. Based on these parameters, different Deep network architectures are compared and a suitable one is selected to implement in the FPGA.

The MNIST (Modified national institute of standards and technology) dataset which is considered as the benchmark for analysing the performance of character recognition is used for the Deep network simulation. The MNIST data set consists of 70000 scanned grey scale images of handwritten digits and its labels. Each digit has 70000 images along with its expected output. A sample of the data set is given below.

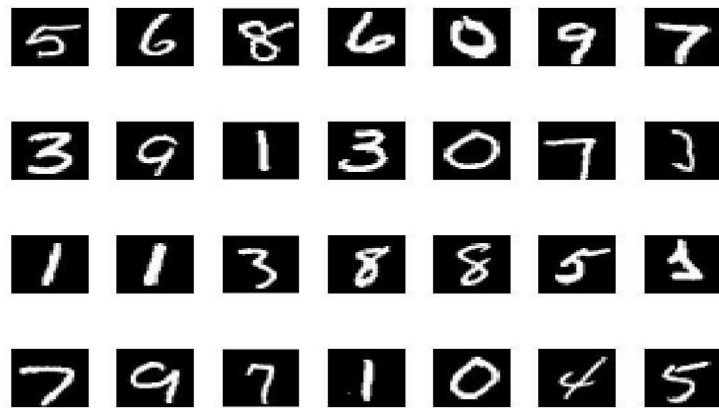


Figure 24 Sample of MNIST data set

The images have been normalised and centred properly to improve the performance of the network. The images are of size 28x28 which makes 784 distinct input data points for each image.

For this particular application, the total number of images is split into 50000 training data, 10000 validation data and 10000 test data. The network is trained with 50000 training data alone. The performance of the system is improved by adjusting network related parameters based on the validation dataset accuracy. The test data is considered to be the unseen data by the network which will be used to measure the performance of the network.

While training the network, the constraints during the implementation on the FPGA also has to be kept in mind. These can be listed as following.

1. Memory limitations.

The memory requirement is due to the number of learnable in the network and due to the storage requirements of largest value of intermediate outputs obtained from each layer. The intermediate results may not be immediately operated on due to hardware limitations and may have to be stored on on-chip memory till the entire operations on that layer is completed. For the later part of this literature, the sum of the above two terms is referred to as the number of data points to be stored.

2. Number of dedicated multipliers available in FPGA

The multiplication operation can be done in parallel or one after another. A higher level of serialism increases the time of operation and the memory requirements while higher level of parallelism will need more no. of multipliers. A trade-off must be reached according to the speed of operation, available hardware and on chip memory. To compare different architectures, it is assumed that for a 'nxn' filter, we need n^2 multipliers. For fully connected layers, the total number of multipliers is the highest number of neurons in a single layer.

3. Total number of slices and flops

This is dependent on the architecture and the data flow implemented in the FPGA. These cannot be determined initially during architecture simulations.

4. Speed of the system

The time at which the output is obtained is directly proportional to the total number of operations that has to be performed during forward propagation. Number of operations is the total number of multiplications and additions done during forward propagation.

Performance of different trained networks are compared using

1. The test set accuracy: The test set is unseen data fed into the network. Total number of correctly predicted outputs is used to calculate the test set accuracy.
2. Average F1 score: F1 score is again a measure of performance of the system based on it recall and precision of its classes.

Precision of a class is the proportion of the identifications which were actually correct among all the positive identifications for a class. It is defined as

$$\text{Precision(P)} = \frac{\text{No. of True positives}}{\text{No. of True positives} + \text{No. of False Positives}}$$

Recall of class is the proportion of the identifications which were correct for a class.

It is defined as

$$\text{Recall}(R) = \frac{\text{No. of True positives}}{\text{No. of True positives} + \text{No. of False Negatives}}$$

Both Precision and Recall is used to analyse the performance of a system. A commonly used term which finds the trade-off between Precision and Recall is the F1 score which is the harmonic mean of the Precision and Recall.

$$\text{F1 score} = \frac{2}{\text{Precision}^{-1} + \text{Recall}^{-1}} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

The average F1 score was calculated as the mean value of F1 scores for all the output classes. For example, in MNIST data set, there are 10 output classes (10 digits). F1 score of each digit is found and averaged to find the mean F1 score.

Better performance of the network can be achieved by increasing the number of filters and the depth of the neural network. As mentioned before, deep networks have shown to have better performance than shallow networks because of superior function approximation capabilities with limited number of filters.

Therefore, there is a trade-off between the performance and the hardware requirements. Different deep network frame works are compared based on the above parameters.

Each of the networks are trained by setting hyper parameters on a trial-and-error basis.

1. Learning rate
2. Decay factor
3. No of epochs
4. Batch size

Adam optimiser mentioned in the CNN basics are used as the optimiser to update the learnable after each iteration.

It has to be noted that these parameters will no way affect the hardware implementation in FPGA. These are the parameters are adjusted during the training to obtain a faster training and higher validation accuracy.

Rather than starting with random architectures, the proven architectures are analysed first. Modifications based on inferences from previous works are incorporated and tested for better

results. Many architectures are tested and tuned to get the best results. The following are some of them which showed best results along with its performance. All of the following architectures are trained on Nvidia GeForce GTX 1050Ti GPU LeNet Architecture.

The LeNet architecture for character recognition based on MNIST data set. It consists of 12 layers with 2 layers of convolutional layers and 3 layers of Fully connected layers. The original LeNet architecture uses the sigmoid function as the activation function. This has been replaced with the superior RELU activation due to vanishing gradient issue of the former.

The average pooling function is used here to decrease the number of features deeper into the network.

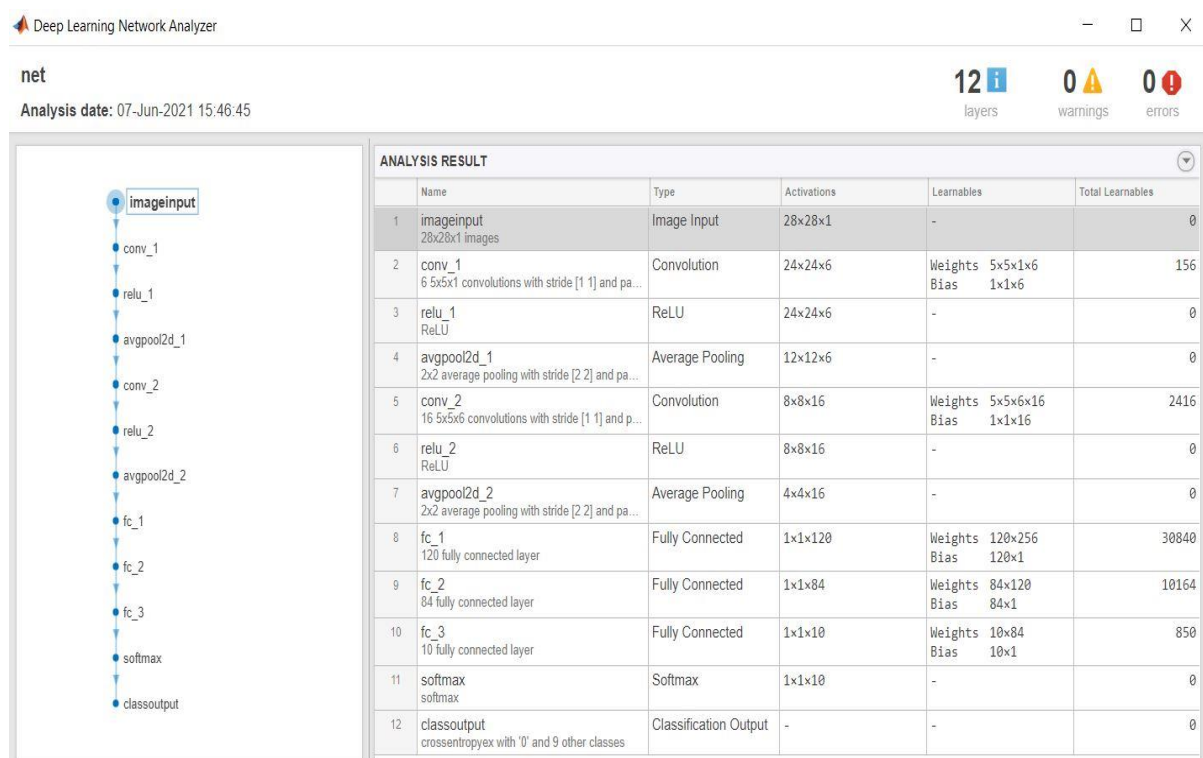


Figure 25 Configuration of LeNet framework

The network was trained with following parameters

Batch size=100, epochs=10, learning rate=3e-4, gradient factor=0.99;

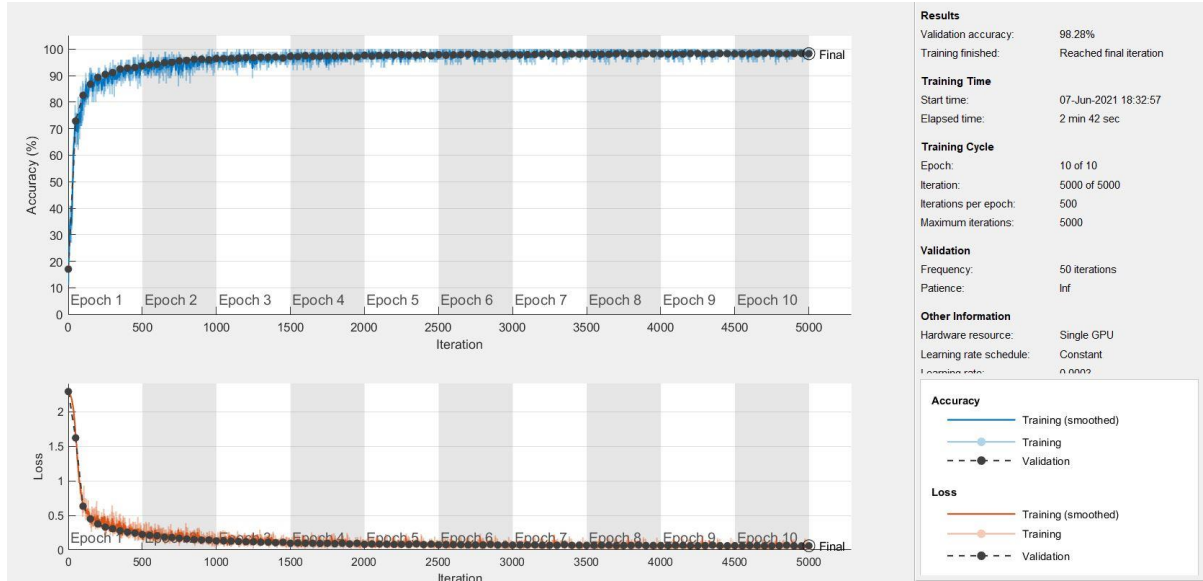


Figure 26 Accuracy and Loss function for the LeNet architecture

The above network gave following results.

- Validation accuracy = 98.28%
- test accuracy = 98.6%
- Average F1 score = 0.98.

From figure 1, FPGA requirements also can be calculated as follows.

1. Number of data points to be stored = Total number of learnable + highest intermediate activations = 47882
2. Total number of Multipliers = 145
3. Total number of operations = 521854

5.1 Network with only Fully connected Layers

The convolutional layers in the above network were completely removed and only FC layers are included and the performance is measured. The FC layer mainly contributes to the number of data points to be stored, thus requiring more on chip memory in FPGA to hold the learnable parameters. The network is trained with a shallow network first and then depth and number of neurons in each layer is increased. Following are the networks and the number of neurons which showed best performance for each depth.

5.1.1 Shallow Network

The shallow network consists of a single input layer and one output layer. There are no hidden layers. The input layer is of size 784 (as input image is 28x28) and output layer of 10 neurons are designed.

The network was trained with following parameters

Batch size=100, epochs=10, learning rate= $3e-2$, gradient factor=0.9;

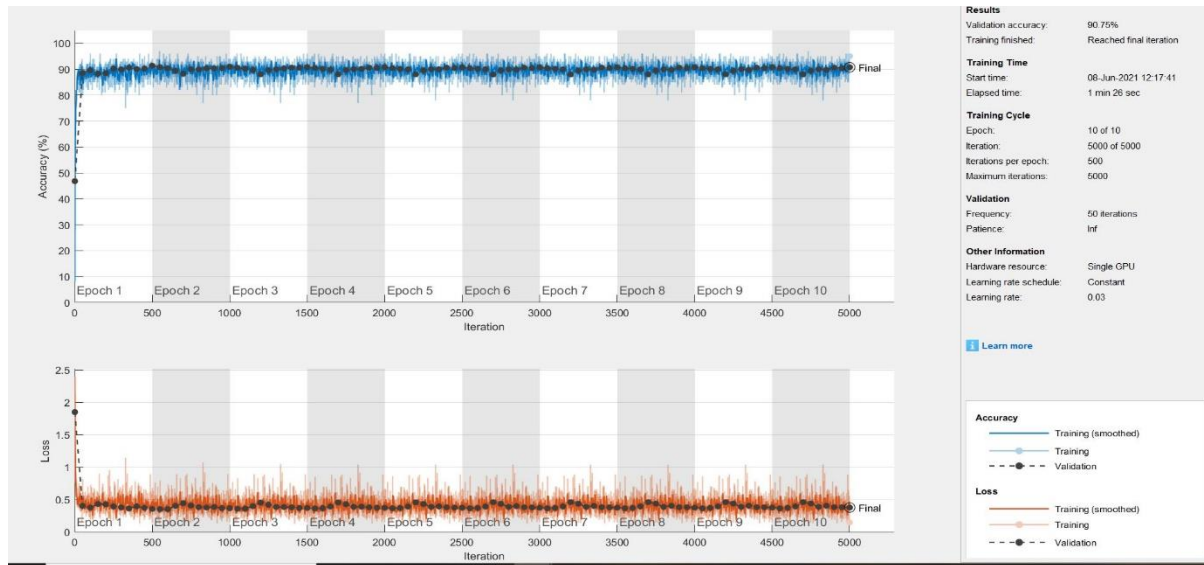


Figure 27 Accuracy and Loss function for the Shallow framework

The performance of the system was as following

- Validation accuracy =90.75%
- test accuracy = 90.65%
- Average F1 score = 0.9
- Number of data points to be stored = 7850
- Total number of Multipliers=10
- Total number of operations=15680

Deep networks with varying number of neurons and depths have been trained. Below are the ones which showed the best results.

5.1.2 FC1

This Network consisted of input layer with 784 neurons, 1st hidden layer with 15 neurons and output layer with 10 neurons.

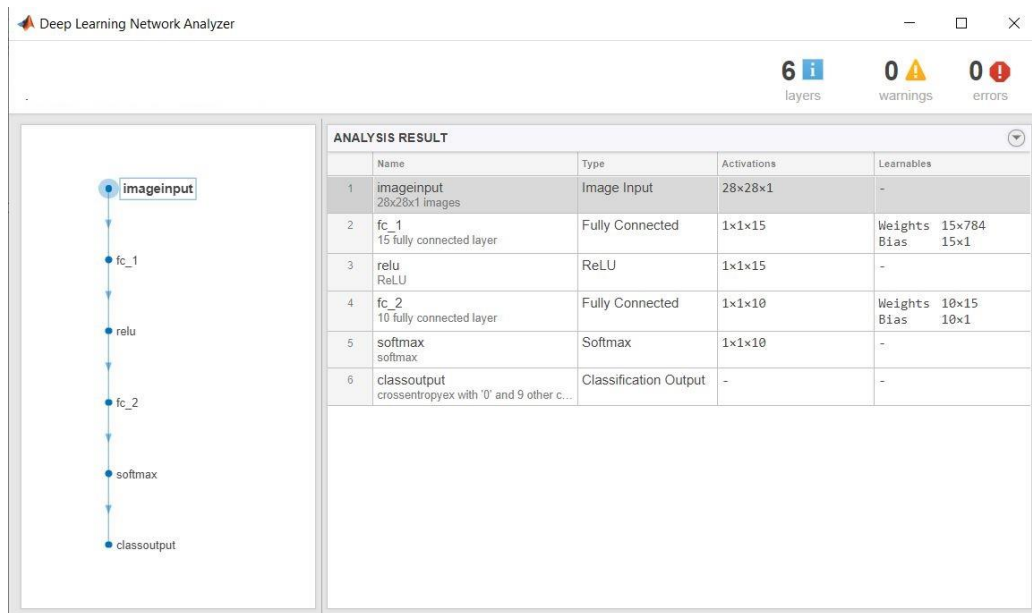


Figure 28 Configuration of FC1 architecture

The network was trained with following parameters

Batch size=100, epochs=10, learning rate=3e-2, gradient factor=0.85;

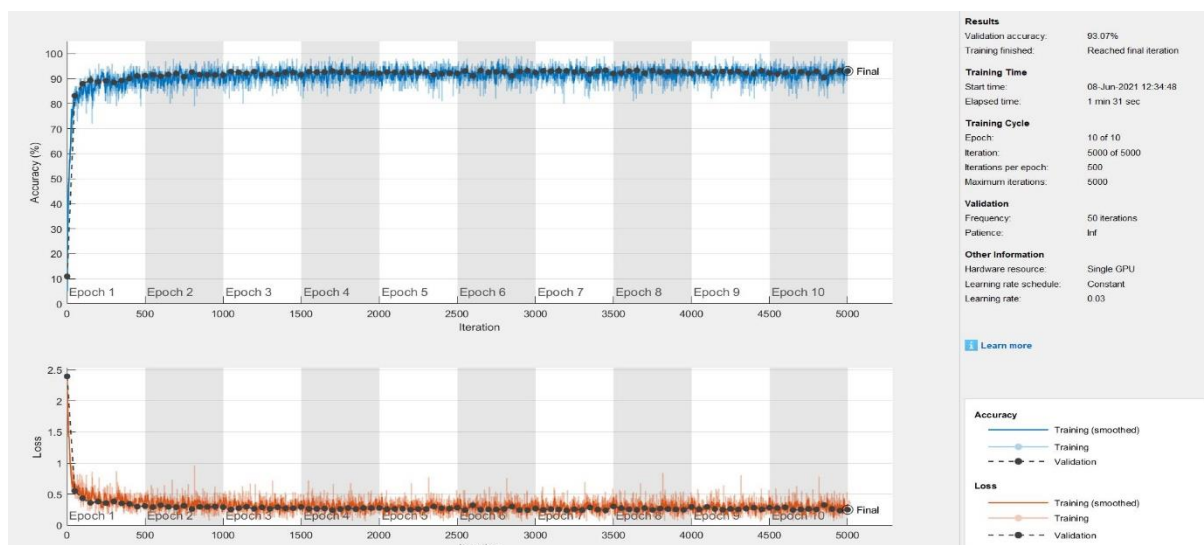


Figure 29 Accuracy and Loss function for the FC1 architecture

The performance of the system was as following

- Validation accuracy =93.07%
- test accuracy = 92.64%
- Average F1 score = 0.92
- Number of data points to be stored = 11950
- Total number of Multipliers=15

- Total number of operations=23820

The accuracy of the system has been found to increase compared to the shallow network but at the cost of more data points and a greater number of operations.

5.1.3 FC2

This Network consisted of input layer with 784 neurons, 1st hidden layer with 45 neurons and output layer with 10 neurons.

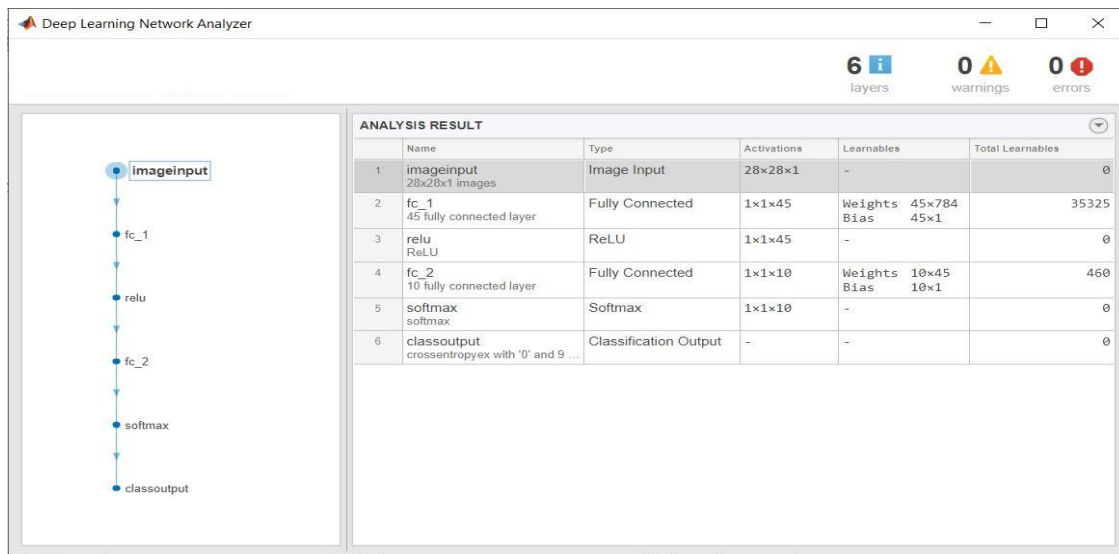


Figure 30 Configuration of FC2 architecture

The network was trained with following parameters

Batch size=100, epochs=10, learning rate=3e-2, gradient factor=0.85;

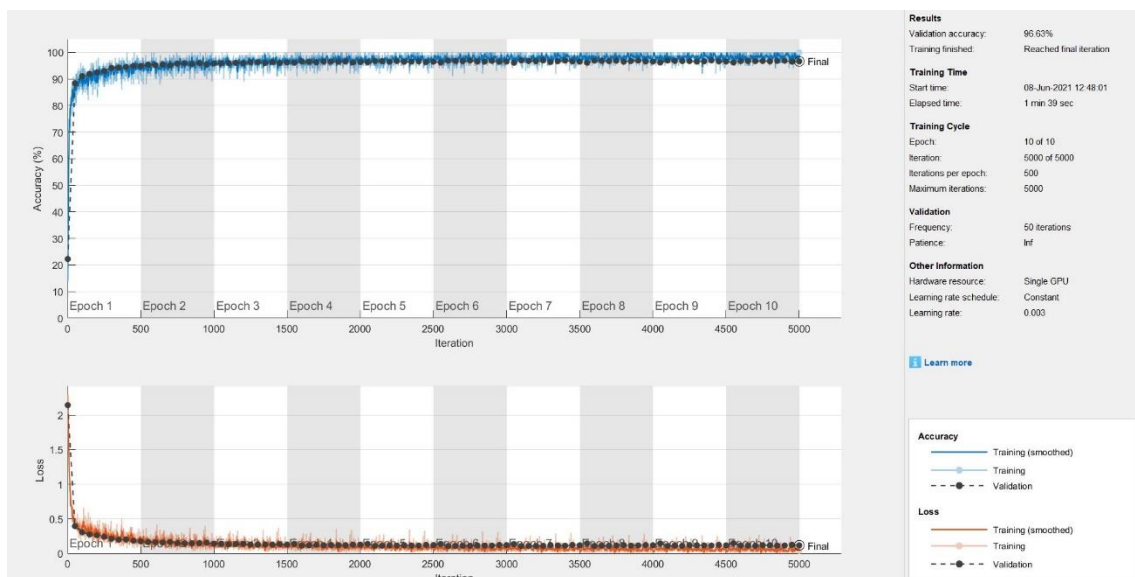


Figure 31 Accuracy and Loss function for the FC2 architecture

The performance of the system was as following

- Validation accuracy =96.63%
- test accuracy = 92.64%
- Average F1 score = 0.92
- Number of data points to be stored = 11950
- Total number of Multipliers=15
- Total number of operations=23820

5.1.4 FC3

This Network consisted of input layer with 784 neurons, 1st hidden layer with 40 neurons, 2nd hidden layer with 20 neurons and output layer with 10 neurons.

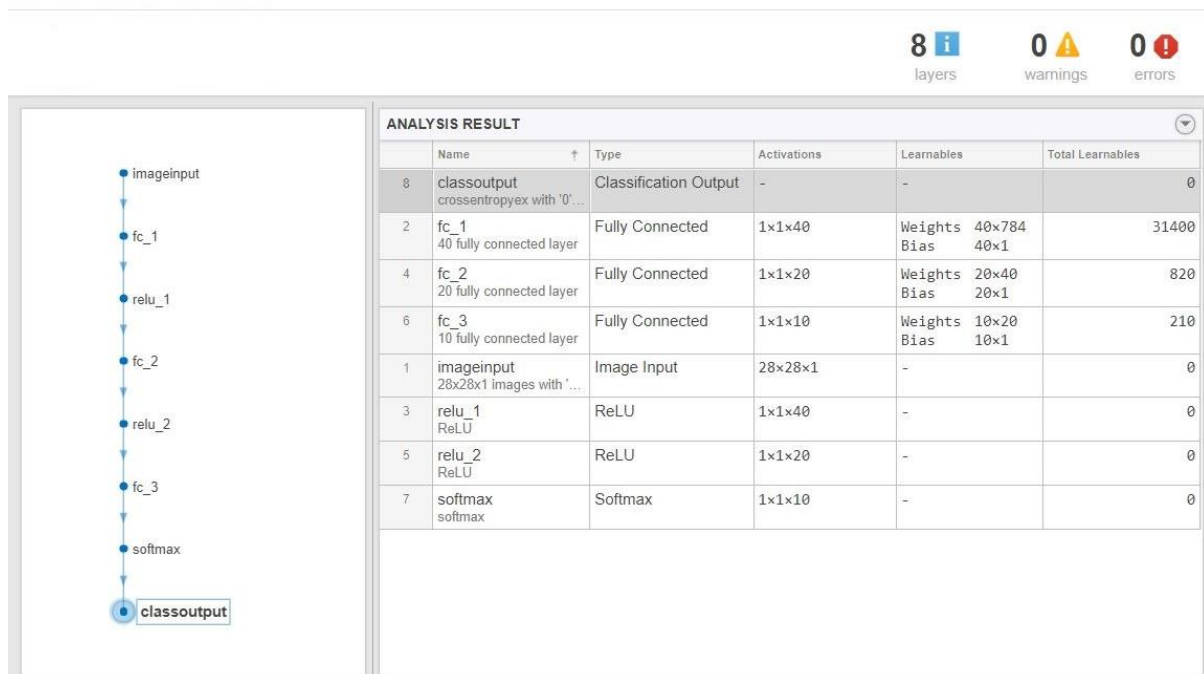


Figure 32 Configuration of FC3 architecture

The network was trained with following parameters

Batch size=100, epochs=10, learning rate=3e-3, gradient factor=0.9;

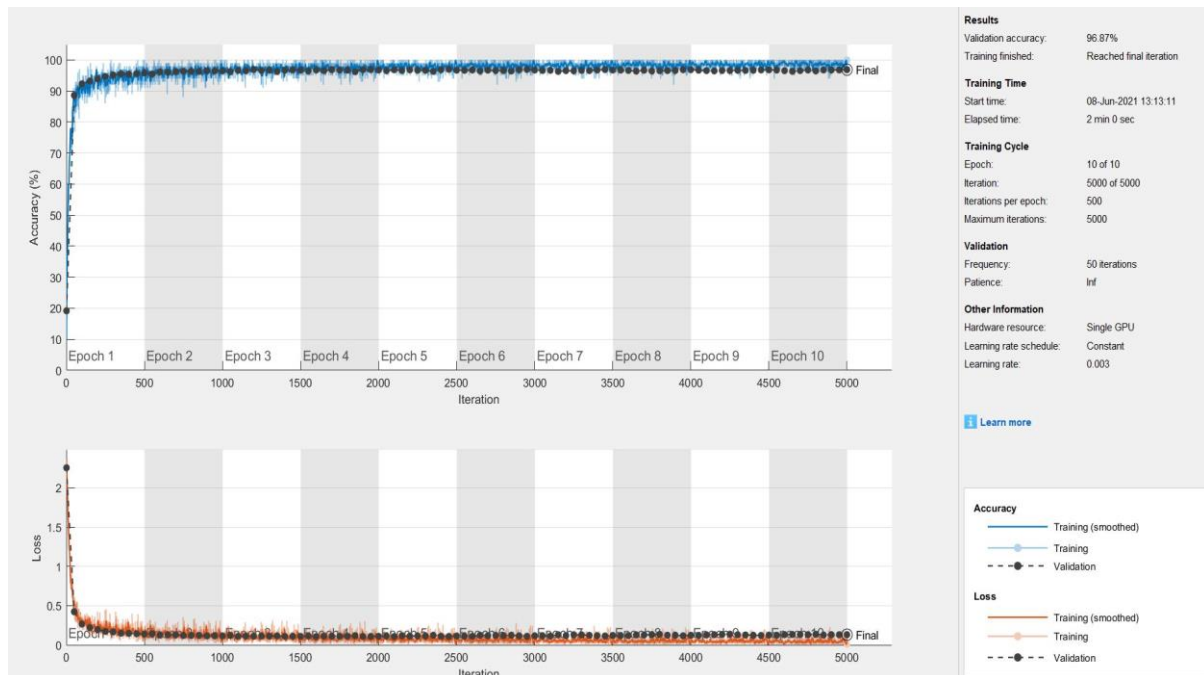


Figure 33 Accuracy and Loss function for the FC3 architecture

The performance of the system was as following

- Validation accuracy =96.87%
- test accuracy = 96.76%
- Average F1 score = 0.967
- Number of data points to be stored =35825
- Total number of Multipliers=40
- Total number of operations=71460

5.1.5 FC4

This Network consisted of input layer with 784 neurons, 1st hidden layer with 25 neurons, 2nd hidden layer with 20 neurons, 3rd hidden layer with 15 neurons and output layer with 10 neurons.

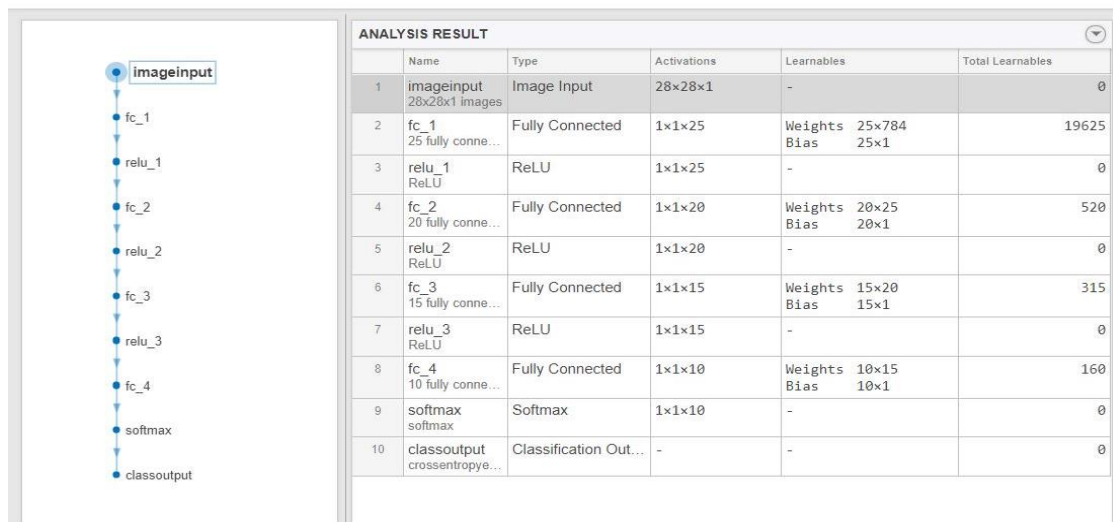


Figure 34 Configuration of FC4 architecture

The network was trained with following parameters

Batch size=100, epochs=10, learning rate=3e-2, gradient factor=0.85;

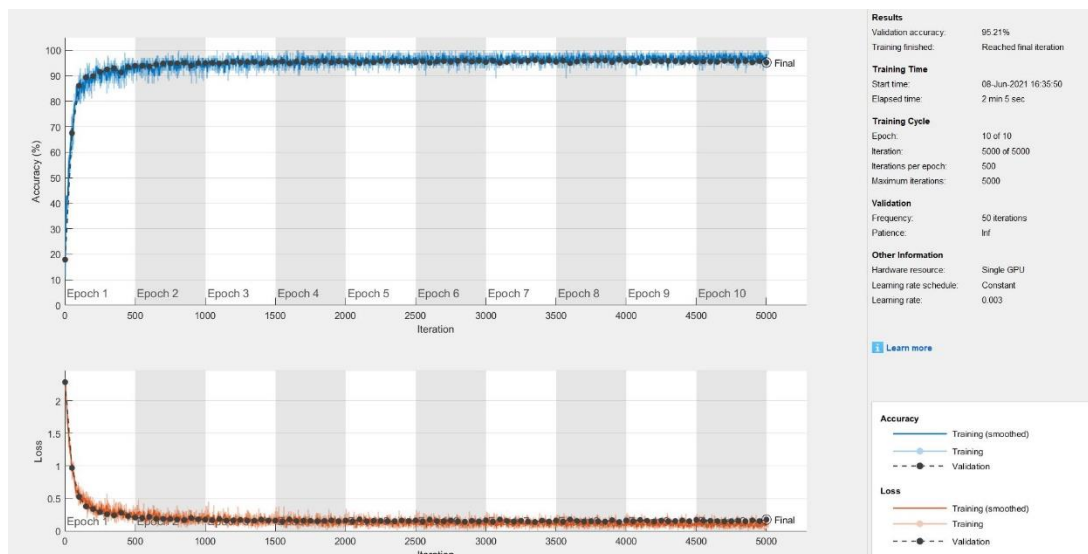


Figure 35 Accuracy and Loss function for the FC4 architecture

The performance of the system was as following

- Validation accuracy =95.21%
- test accuracy = 95.39%
- Average F1 score = 0.95
- Number of data points to be stored = 20645
- Total number of Multipliers=25
- Total number of operations=41170


5.2 Convolutional Layers only

The fully connected layers are removed and the network is trained only with the convolutional layers. The convolutional layers extract the features out from the trained networks and uses it to predict the outputs.

The convolutional layers do not need much memory to store its learnable parameters. Most of the memory requirements come from the intermediate activations. If the network with just convolutional layer gives better performance than the best performing Fully connected network with less memory, the FPGA implementation can be done with just convolutional layers.

5.2.1 CNN-1

This architecture is similar to the configuration in LeNet architecture. The filter sizes are kept of size 5x5 across the layers. Average pooling and Relu non linearity is also used. An extra layer of convolutional layer with 5x5 is added and average pooling with stride of 4 is used to adjust the number of outputs in the output layer to 10 for the classification layer.



ANALYSIS RESULT					
	Name	Type	Activations	Learnables	Total Learnables
1	imageinput 28x28x1 images	Image Input	28x28x1	-	0
2	conv_1 6 5x5x1 convolutions with stride [1 1] ...	Convolution	24x24x6	Weights 5x5x1x6 Bias 1x1x6	156
3	relu_1 ReLU	ReLU	24x24x6	-	0
4	conv_2 6 5x5x6 convolutions with stride [1 1] ...	Convolution	20x20x6	Weights 5x5x6x6 Bias 1x1x6	906
5	relu_2 ReLU	ReLU	20x20x6	-	0
6	avgpool2d 4x4 average pooling with stride [4 4] ...	Average Pooling	5x5x6	-	0
7	conv_3 10 5x5x6 convolutions with stride [1 1] ...	Convolution	1x1x10	Weights 5x5x6x10 Bias 1x1x10	1510
8	softmax softmax	Softmax	1x1x10	-	0
9	classoutput crossentropyex with '0' and 9 other d...	Classification Output	-	-	0

Figure 36 Configuration of CNN-1 architecture

The network was trained with following parameters

Batch size=100, epochs=8, learning rate=3e-4, gradient factor=0.85;

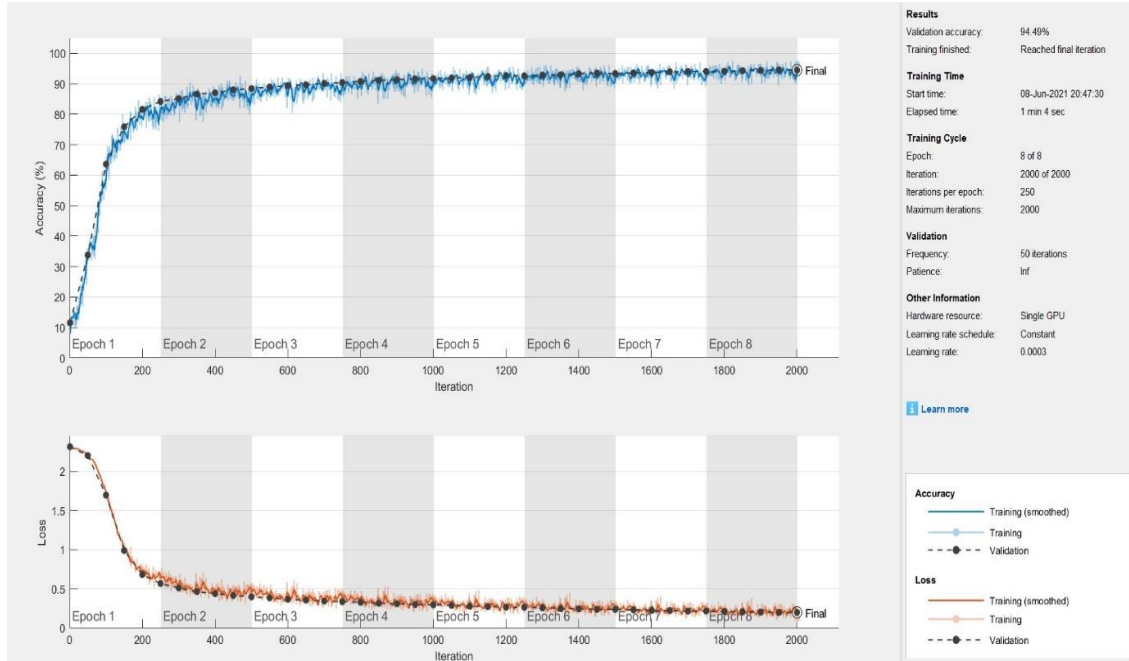


Figure 37 Accuracy and Loss function for the CNN-1 architecture

The performance of the system was as following

- Validation accuracy = 94.49
- test accuracy = 94.33
- Average F1 score = 0.94
- Number of data points to be stored = 6028
- Total number of Multipliers=25
- Total number of operations=895800

5.2.2 CNN2

The convolutional layers in the latest deep networks for image recognition such as Alex net and VGGnet employs small filters of size 3x3 rather than big filters. It was found that this reduces the no. of learnable while keeping the performance of the system same. An intuitive way to look into this is that large filters extracted more background information rather than the actual features. All of the filters in above network were thus replaced with 3x3 filters.

The average pooling is also replaced with max pooling layers of size 2x2 with a stride of 2.

The configuration of the network is as below



Figure 38 Configuration of CNN-2 architecture

The network was trained with following parameters

Batch size=50, epochs=8, learning rate=3e-4, gradient factor=0.9;

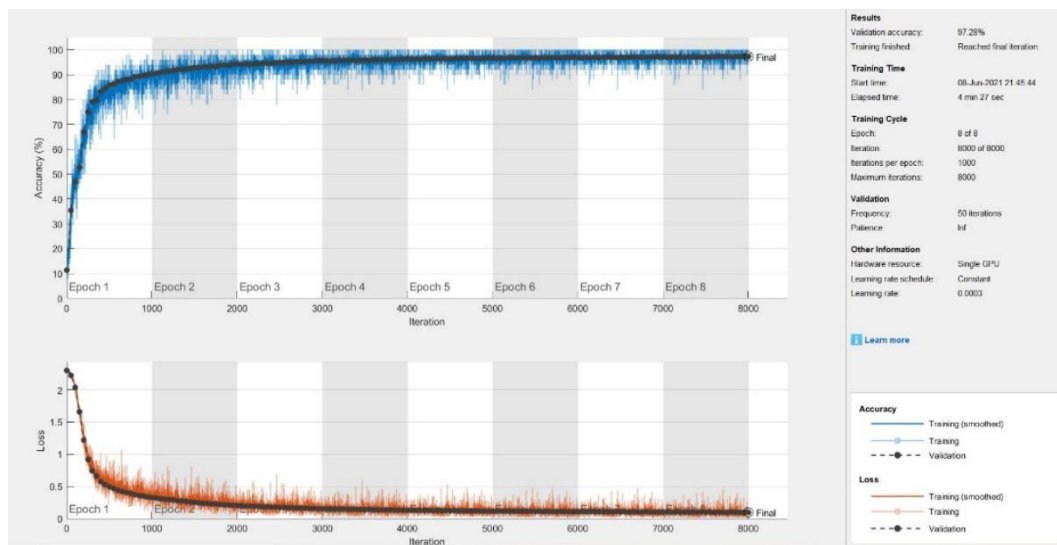


Figure 39 Accuracy and Loss function for the CNN-2 architecture

The performance of the system was as following

- Validation accuracy = 97.28
- Test accuracy = 97.42
- Average F1 score = 0.97
- Number of data points to be stored = 9590

- Total number of Multipliers=9
- Total number of operations=386280

5.3 Comparison Report

Deep networks perform better than the shallow networks in terms of accuracy and average F1 score. But it needs a greater number of learnable to achieve its desired accuracy. The Convolutional neural networks gave better results compared to fully connected networks for image recognition. Also, the CNN has a smaller number of data points which means it can be implemented relatively easier on FPGAs with low on chip memory. This comes at the cost of time of operation which can be improved by an efficient architecture and the capability of massive parallelism of the convolutional operation.

The LeNet architecture which has both FC and convolutional layers showed the best performance but it has huge memory as well time of operation.

The CNN-2 architecture which showed the best result in terms of accuracy is chosen for implementation in the FPGA. It has 1.18% less accuracy compared to LeNet but has reduced the memory requirement by 80% and increased the speed of operation by 25.98%. It also has 0.41% better accuracy compared to the best performing FC network (FC-3) with reduced memory requirement of 70.42% but the speed of the network is reduced by 5.96 times.

Network Configuration	No. of data	Total no. of operations	Validation accuracy	Test accuracy	Average F1 score	No. of multipliers
LeNet	47882	521854	98.28	98.6	0.98	145
Shallow	7850	15680	90.75	90.65	0.90	10
FC-1	11935	23820	93.07	92.64	0.92	15
FC-2	35785	71460	96.63	96.76	0.967	45
FC-3	32430	64790	96.87	96.53	0.964	40
FC-4	20645	41170	95.21	95.39	.95	25
CNN-1	6028	895800	94.49	94.33	.94	25
CNN-2	9590	386280	97.28	97.44	0.97	9

Table 1 Comparison report of different Deep networks

Looking deeper into CNN-2, some of the wrongly classified digits by the network is shown below. These can be confusing even for humans to correctly classify them at first sight. It can be safely assumed that the well written hand digits can be correctly identified by this network with confidence.

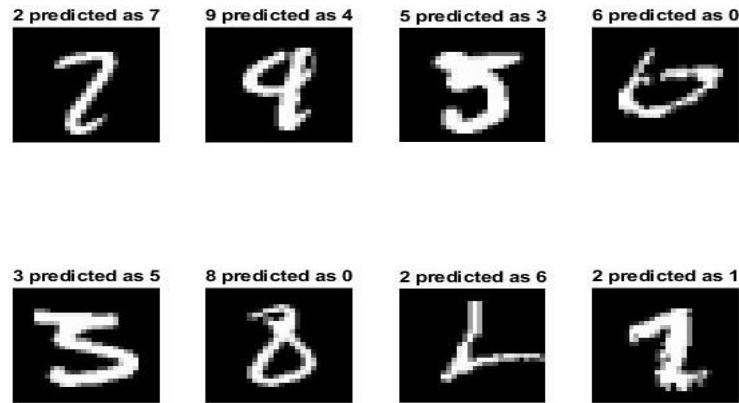


Figure 40 Some of the wrongly predicted images in CNN-2

5.4 Data Quantisation

The Xilinx Spartan 3e XCS3500e FPGA has a block ram of 360 kbits, Block rams are the dedicated memory resources available within to store the results of intermediate operations. All the learnable, intermediated partial outputs and the results are quantised and stored in this memory in 2's complement binary form in a fixed-point format.

The data width for the fixed-point format and the position of the radix point is to be determined. This is determined by analysing the range of values of the weights, biases and the intermediate activations of the trained network. The goal is to find the number of decimal and fractional bits which can estimate the outputs with less degradation in accuracy.

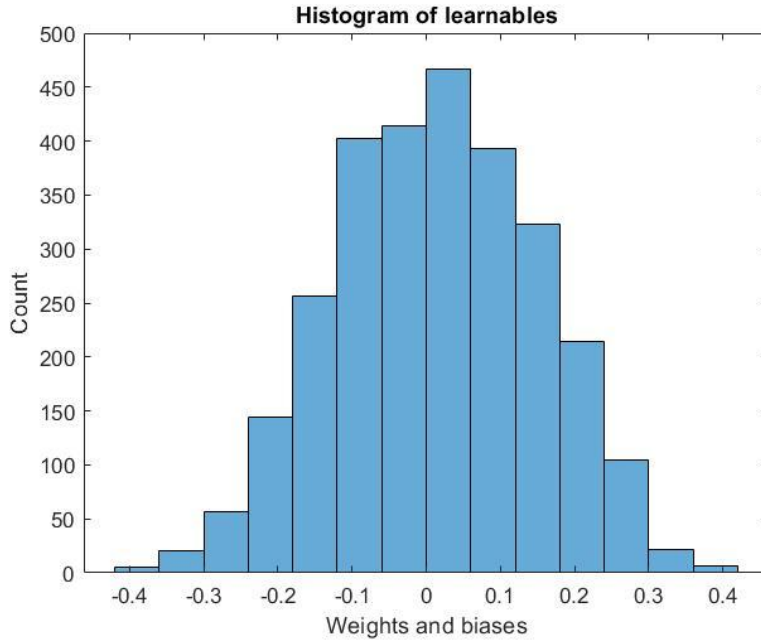


Figure 41 : Histogram of weights and biases of CNN-2

The weights and biases are having the range of values shown in Figure-5.5.1. Most of the weights and bias values lies between the range of -0.2 to 0.2. To resolve this with minimal errors, 7 bits of decimal points. This can resolve numbers up to $2^{-7} = 0.0078125$.

Similar analysis is done for the intermediate activations of all the convolutional layers. The intermediate outputs of the convolutional layers of random 100 input images are analysed. It can be seen that the range of values in the first layer of CNN is concentrated more near -0.5 to 0.5. This means that a greater number of bits are required in the decimal portion to resolve these values.

As we go deeper in the network, the range of values begin to spread out across a wider range with less emphasis on the resolution. The range of values is spread from -25 to 15. It means more decimal bits are required to represent these values. This means we need 5 decimal bits to represent these values accurately.

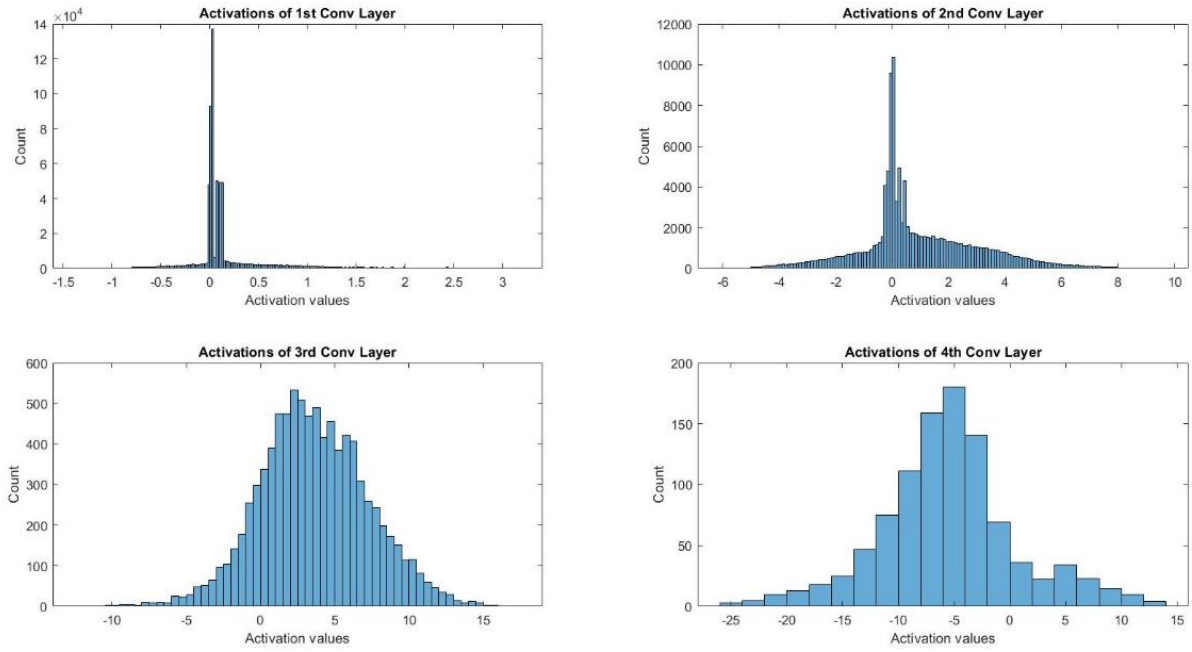


Figure 42 Histogram of activations of 100 random images of each layer

Going with this technique requires 1 sign bit, 5 decimal bits and 7 fraction bits to accurately estimate the activation values. To reduce the total data width from 13, another suitable approach is implemented.

The filter weights and biases of the trained network from each of the layer is scaled by a certain scale factor. Scaling weights and biases of a convolutional layer equally scales the activations obtained from that convolutional layer by that scale factor. This in no way affects the performance of the system but actually helps in representing these values with more accuracy with a smaller number of bits.

The number of bits chosen were as 1 sign bit, 2 decimal bits and 7 fraction bits initially. The maximum and minimum values which can be represented by this configuration is 3.9921875 and -4 respectively. Any value above the maximum value will be saturated to the maximum value and any value below the minimum value will be under flowed to the minimum value. This can make a good fraction of the total activations be wrong values. Propagating these activations through the network will worsen the accuracy drastically. The network was found to have an accuracy of 73.9% with the above quantisation. There is a reduction of 23.54% for the quantised network.

To improve the accuracy, the weights and biases were scaled by scale factors. Let the scale factor of 2nd, 3rd and 4th layers be v_1 , v_2 and v_3 respectively. In order to find the scale factors giving best accuracy, a brute force algorithm which searches through the scale factors was

implemented. Forward propagation results were checked on the 1000 random images from the validation set and following results were obtained.

v3	v2	v1	Accuracy
1	1	1	73.9
1	1	1.25	89
1	1	1.5	94.5
.	.	.	.
1	1.75	1.25	95
1	1.75	1.5	96.1
1	1.75	1.75	96.6
1	1.75	2	96.2
1	1.75	2.25	96.3
.	.	.	.
4	4	3.5	92.3
4	4	3.75	92.2
4	4	4	92.1

Table 2 The accuracy values for various scale factors

By changing the values of the scale factors, the accuracy was seen to improve. This is because of better approximation of the activations with the available number of bits. The scale factors which showed the best results were v1=2, v2=1.75, v3=1.75. The accuracy with these scale factors was 96.6% which is just 0.84% below the original accuracy. The activations after scaling are as below.

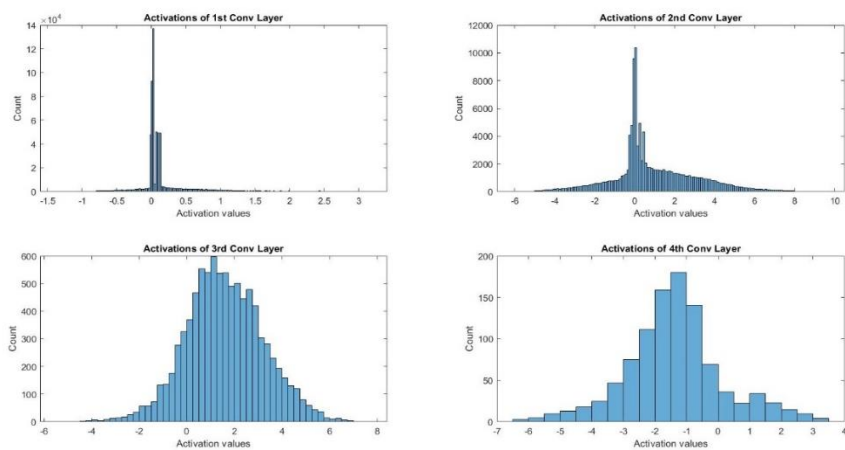


Figure 43 Histogram of activations of 100 random images of each layer after scaling

It can be seen that there are very a smaller number of activations outside the maximum and minimum range.

The similar method was also checked for 8-bit representation to see the extend of the improvement. 1 sign bit, 1 decimal bit and 6 fractional bits were given. The scale of improvement in accuracy was drastic from 46.1% when using default values to 94.5% when the scale factors of $v_1=1$, $v_2=1.5$ $v_3=3$ was used.

5.5 Extension to Devanagari Dataset

A similar convolutional layer only architecture was extended to identify handwritten Devanagari script. Devanagari script consists of 36 consonants and 16 vowels. Each character has a line on top of it which is used to connect characters and form a word. Unlike the English digits, the shapes of this script are complex with much more loops, curves and edges. The database for this script was developed by ISI Kolkata. It consists of 36 Devanagari letters except the vowels. Each letter has a total of 72000 scanned samples.

For training, the sample is divided into 1800 training samples, 100 validation samples and 100 test samples. A convolutional layer-based architecture was trained to predict the letters. The number of training samples per class is 1800 compared to 5000 in MNIST data set. This could affect the accuracy of the network. Previous works were able to obtain a peak of 95% for Devanagari character identification.

A sample from each class is shown below.

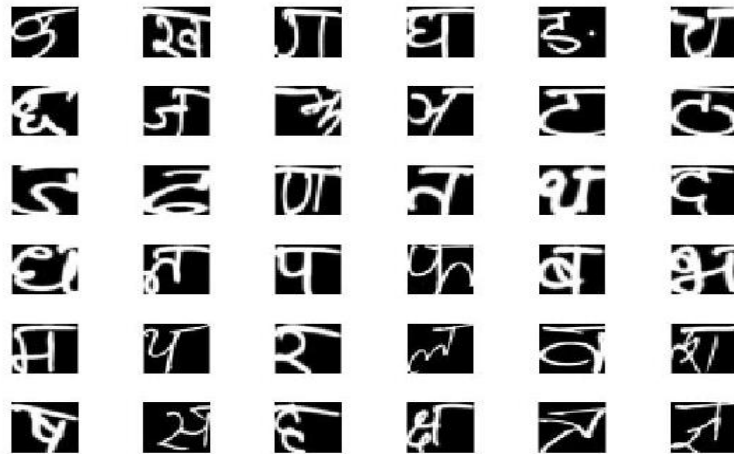


Figure 44 Sample of Devanagari data set

All the convolutional filters used were of size 3x3. The number of channels in each layer is different compared to that of MNIST. The out-channel length of the first layer was again fixed to 10 as the number of activations from this layer is highest and determines BRAM capacity along with number of learnable. In channels less than 10 was found not give good accuracy. The number of neurons in the outer channel is also different as we have 36 different classes for the Devanagari script. The following architecture showed the best result for a convolutional layer only network.



Figure 45 Configuration of Devanagari framework

The network was trained with following parameters.

Batch size=50, epochs=10, learning rate=3e-4, gradient factor=0.9;

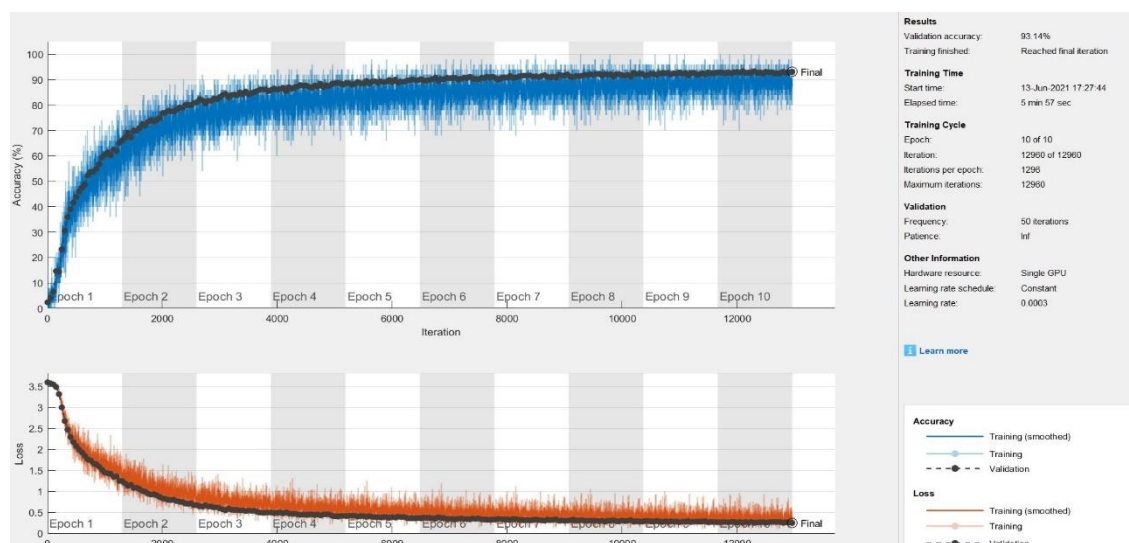


Figure 46 Accuracy and loss function for Devanagari framework

The performance of the system was as following

- Validation accuracy = 93.14
- test accuracy = 90.86
- Average F1 score = 0.9135
- Number of data points to be stored = 14776
- Total number of Multipliers=9
- Total number of operations=603936

5.5.1 Data Quantisation

The similar data quantisation method was used for the Devanagari data set. Using the 3.7 fixed point format without any scale factor gave a test accuracy of 53.72 %. A brute force algorithm to check various scale factors was implemented for this data set. The following result was obtained. Applying scale factors of $(v1, v2, v3) = (2, 1.25, 2)$ gave an accuracy of 84.89%. The peak accuracy could not be improved beyond this point.

v3	v2	v1	Accuracy
1	1	1	53.72222
1.25	1.25	1.25	69.75
.	.	.	.
2	1.25	1.25	69.5
2	1.25	2	84.88889
2	1.25	2.75	82.72222
.	.	.	.
3.5	3.5	2	54.72222
3.5	3.5	2.75	54.72222
3.5	3.5	3.5	49.97222

Table 3 The accuracy values for various scale factors in 3.7 format

Since the 3.7 format was not able to give good accuracy result, the 3.7 format was changed to 3.9 format. Since the multipliers inside the FPGA are of 18 bits, this won't be an issue as far as the FPGA is concerned. The percentage of slice consumption would be increased to accommodate more number bits. The memory requirement also will be increased by a factor of 20%. The 3.9 format gave an accuracy of just 35.77 % without scaling. The results with scale factors were as following.

v3	v2	v1	Accuracy
1	1	1	35.77778
1	1	1.5	59.61111
1	1	2	77.75
.	.	.	.
3.5	1.5	2	86.38889
3.5	1.5	2.5	89.47222
3.5	1.5	3	89.88889
3.5	1.5	3.5	89.38889
3.5	1.5	4	89.11111
.	.	.	.
4	4	3	86.16667
4	4	3.5	84.38889
4	4	4	82.94444

Table 4 The accuracy values for various scale factors in 3.9 format

The scale factors of (v3, v2, v1)= (3.5,1.5,3) were used while implementing in the FPGA.

5.6 Summary

The parameters which measure the performance of the network was defined at the beginning of the chapter. The parameters that have to be kept in mind during the FPGA implementation was also defined. After this, various deep network architectures were trained and these parameters were identified for MNIST data set.

It was seen that the performance of the shallow fully connected network was the least. It was seen that the performance of the deep networks increased with increase in depth but at the cost of memory and computational time. Convolutional networks gave good performance with limited memory requirement but at the cost of more computational time. The best results were achieved when both convolutional and fully connected layers were used.

Based on these results, suitable architectures for character recognition of MNIST and Devanagari script was chosen.

6 VLSI ARCHITECTURE DESIGN FOR FPGA

In this chapter, the VLSI architecture implemented in the FPGA will be discussed. A comparison between non-systolic architecture-based design and improvement in performance will also be discussed. Finally, the design of systolic based architecture will be discussed in depth along with the simulation results.

Different elements constituting FPGA board and its specifications is explained briefly below so that hardware utilization of architecture can be analysed and compared.

6.1 Xilinx Spartan-3e FPGA Board

Spartan 3E family of FPGA are designed so as to suite for cost effective design applications. Architecture of FPGA board consists of 5 fundamental programmable functional elements, CLBs, IOBs, BRAM, Multiplier blocks and DCM blocks. Spartan-3e XC3S500 FPGA has 4650 slices, 9312 slice flipflops and 4 input LUTs and 66 bonded IOBs.

Configurable Logic Blocks (CLBs) has Look Up Tables (LUTs) to perform logic functions and Flipflops that act as storage element. It functions as logic device for wide variety of logical functions. Input Output Block (IOBs) control data flow between input/output pins and internal logic device. Digital Clock Manager (DCM) provide perform distribution, delaying, multiplying, dividing, phase shifting etc of clock signals. Each functional element has associated switch matrix that controls multiple connections to routing. FPGA is programmed by loading configuration data to static CMOS Configuration Latches (CCLs). It is stored externally in non-volatile memory such as PROM.

The Spartan-3e XC3S500 has 20 dedicated 18x18 multipliers which helps to implement fast and efficient arithmetic function (multiplication) with minimal use of general-purpose resources. Both signed and unsigned multiplication of 18-bit inputs can be performed with multiplier. Since multiplier blocks are located adjacent to RAM blocks and share routing resources with them, access of memory for multiplication can be efficiently utilised.

6.2 Storage Elements

6.2.1 Distributed RAM

LUTs in the FPGA can be programmed as Distributed RAM. As the name suggests small blocks of memory will be synthesized at different parts of board. It is useful for the applications

where small amounts of memories have to be associated with different portions of hardware modules.

6.2.2 Block RAM

Spartan 3e FPGA contain 360Kb BRAM which is organized as 20 blocks of 18 kb memories. These block ram memories offer fast and flexible storage of large amounts of on chip data. Read and write operations are synchronous to clock. It can be configured to function as single port or dual port RAM. In dual port values stored in two different addresses can be accessed simultaneously with read and write enable signals associated with each of the access. A name of the form RAMB16_S9_S8 is identified as dual port with 9 and 8 width data ports. Similarly, single port can be identified with RAMB16_S8 if data port width is 8.

6.2.3 Memory for CNN architecture

As large blocks of data are required to be stored in the handwritten character inference task, Block RAM is preferred over distributed RAM. Since distributed RAM uses LUTs, availability of resources for logic design might be less if distributed RAM is used. In the below figure, schematic and resource utilization of Memory that stores input features for character recognition named as ‘infeature_memory’ is given. The module is synthesized as dual port RAM. As maximum size for feature storage is in the first layer (26 X 26 X10), input feature and output feature memories are allocated 8000 with address width of 13 and datawidth of 10.

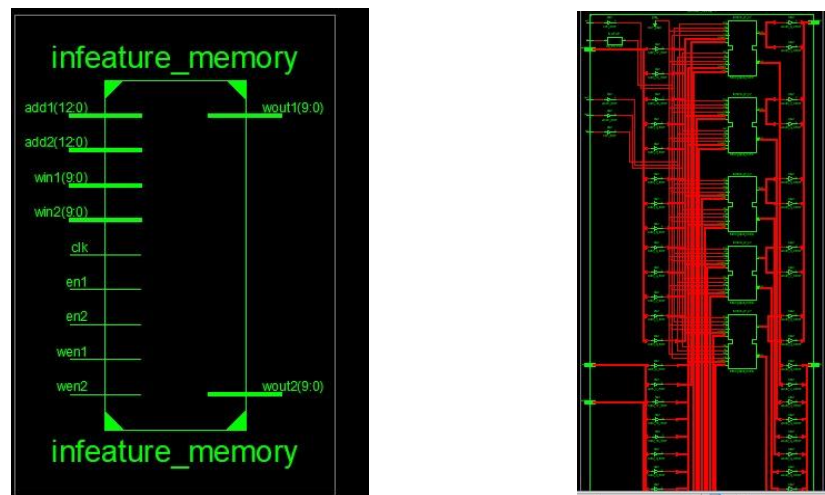


Figure 47 Schematic of input feature memory synthesized as dual port RAM

Memory module is synthesized with 5 BRAM blocks of RAMB16S_2S2 each as shown below. It means that each data port of one block RAM is of width 2 and hence with 13 address lines

$2^{13} \times 2 = 16 \text{ kb}$ is possible. Since one block is of size 18 kb this corresponds to one block of RAM.



Figure 48 Single BRAM in memory module

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	0	4656	0%	
Number of bonded IOBs	71	66	107%	
Number of BRAMs	5	20	25%	
Number of GCLKs	1	24	4%	

Figure 49 Resource allocation

Similarly, output feature map needs another 5 BRAM for storage. Hence remaining 10 BRAMs can be allocated for Weights and biases storage if datawidth of quantization is 10.

6.3 Overall Architecture

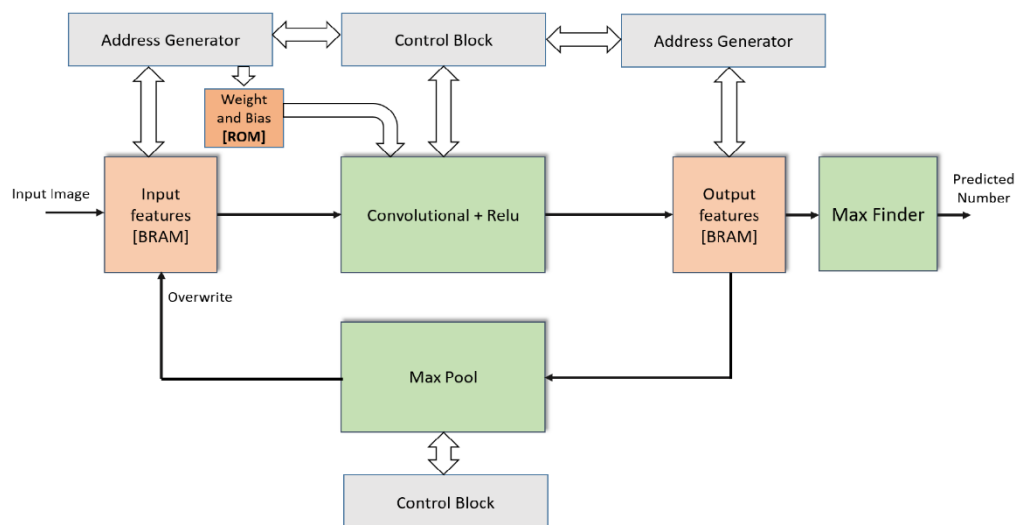


Figure 50 Overall implemented architecture in FPGA

The above figure represents the basic architecture which is implemented in the FPGA. It mainly consists of two functional blocks namely convolutional block and Maxpool block.

The input image is read from the external memory directly into the Block RAM of the FPGA. The BRAM is divided into two parts to store the input features and the output features obtained after convolving the input features with the convolutional filters.

The weights and biases are also stored in the BRAM and is configured as Read only memory. The necessary inputs are read per requirement with the help of an address generator and control block and is fed into the convolutional layer. The outputs obtained from the convolution block is stored into the output features. Once the entire convolutional operations are finished, the control block enables the Maxpool block and disables the convolutional block.

The output features serve as the input to Maxpool block. The output from the Maxpool block is saved and overwritten in the input feature BRAM as these values are not required anymore for the further calculation. The reuse of memory is required because of the limited memory resources available to store the intermediate features extracted by the convolutional layer. Once the entire maxpooling is finished, the convolutional block is enabled back and Maxpool block is disabled. The process repeats depending on the CNN architecture till the end of iteration.

The final layer in the CNN architecture is a SoftMax layer which is used during the training of the neural network. Implementing a nonlinear function in FPGAs are done with the help of LUTs. But this again consumes extra hardware. Even though SoftMax layers are useful in training the network, it can be replaced with a max finding layer in the forward propagation to find the predicted output.

The input image, the intermediate features and the filter weights are 2D inputs. As the BRAM of the Xilinx FPGAs support address in 1 dimension. All the features are those converted from 2D format to 1D format and is stored inside the FPGA. The address generator generates the index variables which is used to access the data as per required.

6.4 Non-systolic architecture for the Convolutional Layer

An architecture where a tree of multipliers and adders connected together to perform convolution is shown below.



Figure 51 A Non systolic architecture for convolution

In the above example, a 2D convolution is performed between a 3×3 input and 3×3 filter. Since 9 multiplications are done parallelly, 3×3 convolution result will be available immediately whereas 9 clock cycles are required if only one MAC unit is present. Hence single 3D convolution output can be obtained when this process is repeated number of times corresponding to size of input channel length. Although this architecture uses 9 multipliers, its performance is restricted by memory bandwidth. For each computation, 9 different inputs and 9 different filters are required. As all inputs and filters can't be read in single clock cycle, each computation can happen only after specific number of clock cycles (after reading 9 inputs and filters). In convolution, each input data and filter value are required to be used multiple times. Since this architecture does not make use of reutilizing inputs and weights after reading once, memory cost of reading inputs and filters still remains same. So, an architecture which makes use of reutilizing input data and filter values and also reducing partial sum storage is required. In that way memory read and write operations can be reduced along with parallelism and better performance can be obtained. Systolic architecture suits better for such applications.

6.5 Systolic Architecture

The systolic architecture consists of a set of interconnected PEs which performs a simple operation in the input data.

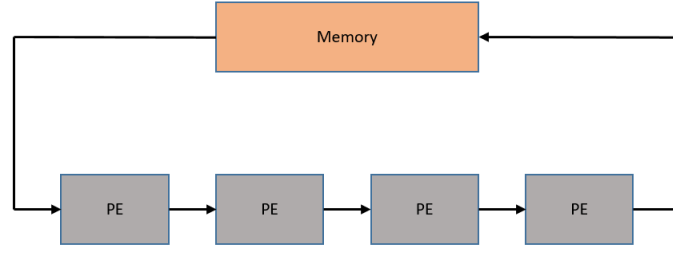


Figure 52 Systolic architecture PE structure

The communication with the external memory takes place only in the boundary cells. After the data has been read from the memory, it passed through several PEs i.e., reused multiple times. The flow of data can be linear, two dimensional or in any other form depending on the application. Usage of multiple PEs along with the data reuse increases the throughput of the system without increasing the memory bandwidth.

Another key feature of systolic architecture is the design of simple and regular PEs. The performance of the architecture therefore can be adjusted by changing the number of PEs proportionally. This makes designing the custom architecture based on systolic system easy to scale based on the hardware availability.

6.5.1 2D convolution using systolic Design

Consider the convolution between the input x and the filter f .

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nn} \end{bmatrix} \quad f = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$$

A single cell designed as follows contains the filter values preloaded into it. Each PE has two inputs namely X_{in} and Y_{in} . X is the current input inside the PE. The outputs are calculated at each cycle as

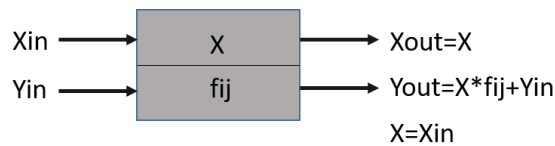


Figure 53 Function of a simple PE

A set of such simple cells are combined to form kernel cell as follows. Five rows of the input are fed into the kernel cell with the scheduling shown below. At $t=0$, the set of inputs x_{11} , x_{12} , x_{13} enters the first kernel. The weighted outputs of the inputs with the corresponding filter value stored in the cell is obtained as the output from the adder.

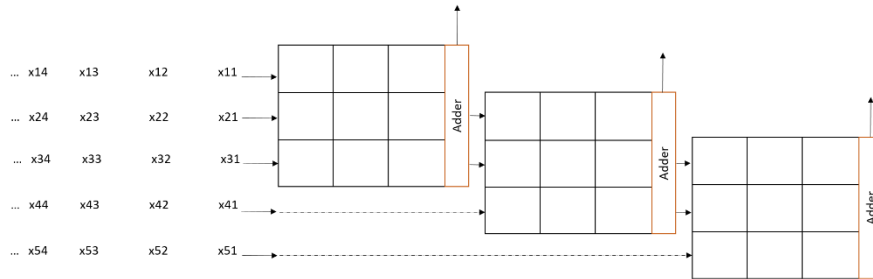


Figure 54 Dataflow in 2D convolution using systolic architecture

The first output corresponding to the first row of the output is obtained after the 3rd cycle from the first kernel. The outputs of the second row of the outputs are obtained from the second kernel and so on. The input x_{21} , x_{31} , x_{41} reaches the second kernel after the 4th cycle. The entire submatrix required to obtain the first output from the second kernel will be available after the 6th cycle. For the above example the entire first three rows of the output are obtained after $n+9$ cycles.

After this the outputs corresponding to 4th, 5th and 6th row can be computed by feeding the inputs from the 4th row to 8th row and so on till we reach the end of the input signal.

The number of kernels can be changed according to the availability of hardware with necessary changes in the scheduling.

For computing 4d convolutions for the CNN, 3 levels of parallelism can be implemented.

1. **Kernel level parallelism-** Multiple kernels compute the 2d convolutions independent as shown above. Increasing the number of kernels also increases the number of data points to be accessed at each time by the same amount.
2. **Input channel parallelism-** The pattern implemented for 2d convolution is replicated along the third dimension (along input channels). There is a need for reading the new set of inputs as well the filter values. For example, consider a convolution between an input of size $5 \times 5 \times 10$ and filter of $3 \times 3 \times 10$. If we replicated the above PE pattern 10 times along the input channel, the number of inputs which

has to read from memory at a time period also increases by 10 times. This could become a bottleneck in terms of bandwidth if increased drastically.

3. **Output channel parallelism-** Replicating the PEs along the output channels. The same input read at an instant can be fed along the output channels. Only the filter values stored inside the PEs change as we move along output channel.

6.6 Systolic Architecture for convolutional layer

Design and implementation details of hardware modules for convolutional layer part of the network is examined below.

6.6.1 Single PE

20 dedicated 18x18 multipliers in Spartan-3e XC3S500 can be used to create the basic PE required. A PE computes the output as shown below for a particular time instance. Once a X_{in} read from the BRAM, it is stored in temporary registers near the PE and is fed into the PE as per requirement.

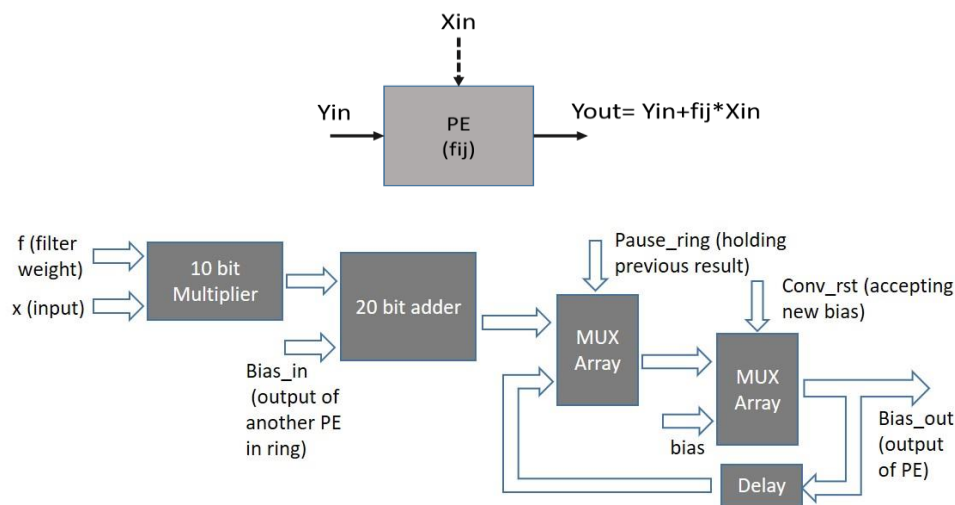


Figure 55 Single PE module

MAC is cascaded with two extra selection MUX to accept Bias value and to hold the previous output. PE is initialised to Bias value as output when a reset ('Conv_rst' in figure) is enabled. This happens at start of each 3D convolution to add bias to convolution of filters and inputs. When another control signal ('Pause_ring' in figure) is enabled PE hold onto the obtained result. This helps when 2D convolution results are obtained in lesser number of cycles than the next set of inputs arrival time. If both control signals are disabled, PE add multiplication result of filter 'f' and input 'x' to 'Bias_in' and gives output.

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	22	4656	0%	
Number of 4 input LUTs	41	9312	0%	
Number of bonded IOBs	83	66	125%	
Number of MULT18X18SIOs	1	20	5%	
Number of GCLKs	1	24	4%	

Figure 56 Resource utilization for single PE

6.6.2 PE Array

All the filters used in the CNN architecture is of size 3x3. So, a total of 9 multipliers are used to make a cell of PEs as shown below.

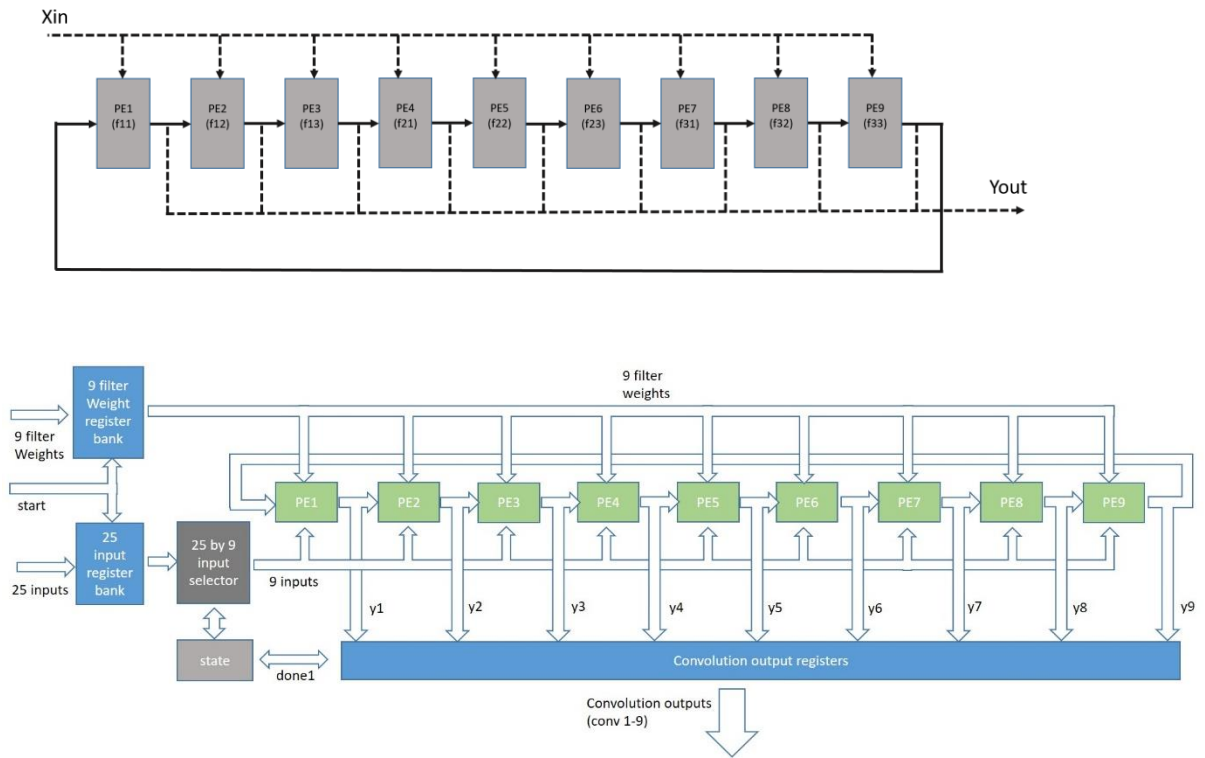


Figure 57 conv PE array module

We can obtain a maximum of 9 outputs after 9 clock cycles from the PEs. So, a 5x5 submatrix is called from the input in every 9 clock cycles. And the result of convolution between 5x5 input and 3x3 filter will be obtained at 9 PE. The scheduling of in the inputs for the first 9 clock cycles is as shown below.

Filter weights	T=1	T=2	T=3	T=4	T=5	T=6	T=7	T=8	T=9
PE-1(f11)	X _{11k}	X _{33k}	X _{32k}	X _{31k}	X _{23k}	X _{22k}	X _{21k}	X _{13k}	X _{12k}
PE-2(f12)	X _{13k}	X _{12k}	X _{34k}	X _{33k}	X _{32k}	X _{24k}	X _{231k}	X _{22k}	X _{14k}
PE-3(f13)	X _{15k}	X _{14k}	X _{13k}	X _{35k}	X _{34k}	X _{33k}	X _{25k}	X _{24k}	X _{23k}
PE-4(f21)	X _{31k}	X _{23k}	X _{22k}	X _{21k}	X _{43k}	X _{42k}	X _{41k}	X _{33k}	X _{32k}
PE-5(f22)	X _{33k}	X _{32k}	X _{24k}	X _{23k}	X _{22k}	X _{44k}	X _{43k}	X _{42k}	X _{34k}
PE-6(f23)	X _{35k}	X _{34k}	X _{33k}	X _{25k}	X _{24k}	X _{23k}	X _{45k}	X _{44k}	X _{43k}
PE-7(f31)	X _{51k}	X _{43k}	X _{42k}	X _{41k}	X _{33k}	X _{32k}	X _{31k}	X _{53k}	X _{52k}
PE-8(f32)	X _{53k}	X _{52k}	X _{44k}	X _{43k}	X _{42k}	X _{34k}	X _{33k}	X _{32k}	X _{54k}
PE-9(f33)	X _{55k}	X _{54k}	X _{53k}	X _{45k}	X _{44k}	X _{43k}	X _{35k}	X _{34k}	X _{33k}

Table 5 Scheduling of the inputs inside the PE

Since BRAM support dual port access and 25 inputs have to be read from memory, 13 cycles are required to read first set of 25 input values. After 13 cycles next set of inputs and filters are fed into registers. PE ring holds onto previous value from 9 to 13 cycles when module has to wait for next set of inputs. This process is repeated until input channel length is reached at which 9 3D convolution results are obtained. After result is obtained, 5x5 window slides by 3 units and next submatrix is called and the process is repeated.

It can be seen that for the first convolution phase, the input read from the rows 1 to 5 and columns 1 to 5 of x input. After the 9 results are obtained, the set of inputs needed are the rows 4 to 8 and columns 1 to 5.

This architecture enables to perform 9 3D convolutions parallelly without need of large temporary registers to store intermediate 2D convolution partial results. Each filter value is reused 9 number of times for 9 clock cycles. Input is also reused after storing in temporary register. 2 such PE arrays are designed parallelly with same inputs broadcasting to both of them.

If n_1 is the row and column size of output feature map for one convolutional layer, C_{in} is the input channel length, C_{out} is the input channel length with filters of size 3×3

Time of computation for one convolutional layer with one MAC and no parallelism

$$= n_1 \times n_1 \times C_{in} \times C_{out} \times 3 \times 3$$

With p PE arrays of with each having 9 PEs as shown above,

$$T_{conv} = \left\lceil \frac{n_1}{3} \right\rceil \times \left\lceil \frac{n_1}{3} \right\rceil \times C_{in} \times \left\lceil \frac{C_{out}}{p} \right\rceil$$

$p=2$ is used in the implemented architecture so that 18 out of 20 available multipliers are utilized.

Device Utilization Summary (estimated values)				[1]
Logic Utilization	Used	Available	Utilization	
Number of Slices	635	4656	13%	
Number of Slice Flip Flops	711	9312	7%	
Number of 4 input LUTs	994	9312	10%	
Number of bonded IOBs	545	66	825%	
Number of MULT18X18SIOs	9	20	45%	
Number of GCLKs	1	24	4%	

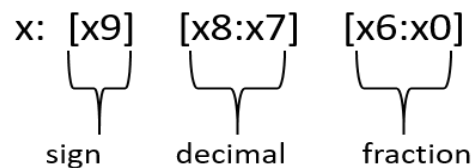
Figure 58 Resource utilization of one convolution PE array module

6.6.3 RELU

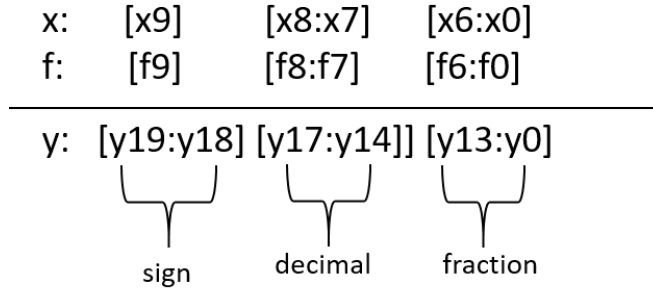
The results of convolution are given to ‘RELU+Truncate’ hardware. The RELU operation is implemented simply by checking the MSB of the final result. The value is written as it is into the BRAM if the value of MSB is 0. Else the value 0 is written. This is implemented easily in the FPGAs with a set of LUTs corresponding to a number of MUX array.

On multiplying 2 numbers (filter values and inputs) of the size x , the output will be of the size $2x$ from the hardware. Adding such $2x$ sized outputs across the in-channels further increases the size. For example, in MNIST data set, the filter, input and bias values are set to be of size 10 bits. Let the number of in channels be 10. The output obtained after the 3d convolution can be take up to size of 20 bits. This value has to be truncated back to the original 10 bits by compression.

The 10-bit data is in 2’s complement with MSB for the sign bit, the next 2 bits for the decimal part and the next 7 bits for the fraction part.



On multiplying such two numbers following is the configuration for the output.



The result has two sign bits of same value depending on the sign of the output. Out of this either one can be used. The 2 LSB 9(y15, y14) is selected out of the 4 decimal bits for the outputs. The first 7 MSB bits (y13: y7) of the fractional part is selected for the decimal part of the output. If the sign bit is 0 (i.e., the output is positive), and if either of the 2 MSB bits of the decimal part (y17 and y16) is set to 1, it means that the output has overflown. In this case, the output is saturated to maximum value. A similar method is used to take care of the under flowing of the output also. This is implemented with a number of MUX arrays.

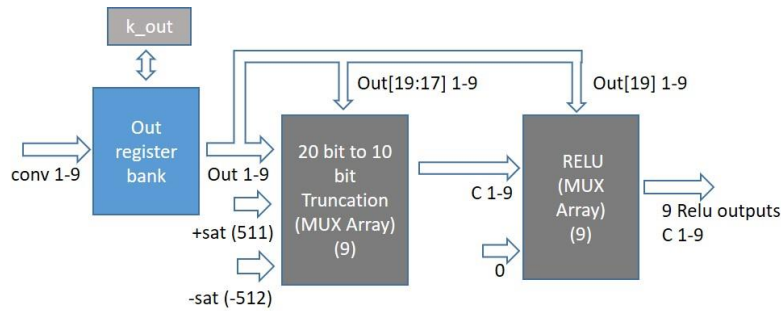


Figure 59 RELU+ Truncate components

After obtaining result of Relu, it has to be written in ‘outfeatures’ memory. Control blocks involved in the convolution and Relu operation and their interconnection to convolution module and memory modules are shown below.

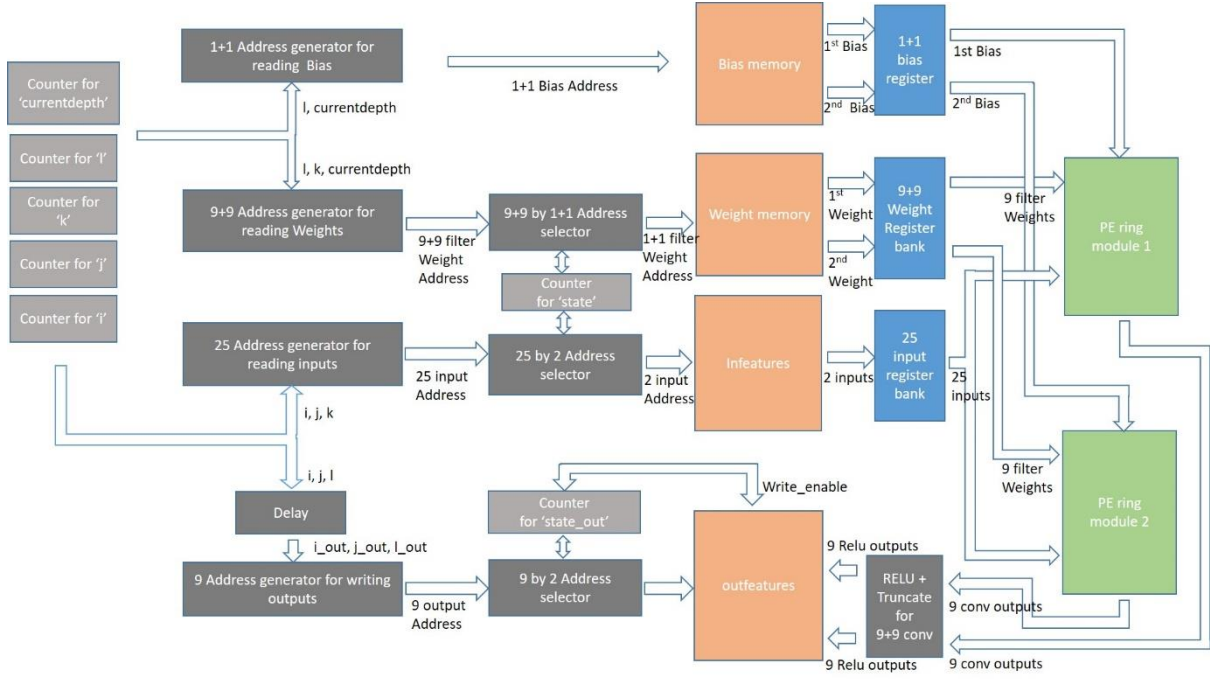


Figure 60 Block diagram for convolutional layer

When a control signal ('Maxpool_on') is disabled, convolution modules perform convolution. A set of counters are used to increment address indices for Infeatures, outfeatures, Weight memory and Bias memory blocks. Infeatures memory is initialized with input image. Input data values are read from Infeatures using the address generated with address generation components. Address generation components convert the 3D address indices to 1D index for the BRAM. In address generation, when multiplication by maximum dimension along row, column etc was required, it can be done either by using shifting operation or by using small multiplier modules. 2 Address values given to memory in each cycle is determined by selector components (MUX array). The values read from Infeatures and Weights are stored in a temporary register bank and are given to convolution module after reading 25 inputs and 9 filters. This allows reading of 25 inputs and 9 filters to happen parallelly while convolution for previously read inputs are performed. After 9 convolution results are ready, they are stored in outfeatures as per the generated address. Memory writing process in outfeatures also happens parallel to convolution and memory reading process from Infeatures.

6.7 Max Pooling

After convolution and Relu is finished, result is read fed to Maxpool layer. Control blocks for Max pooling operation is shown below.

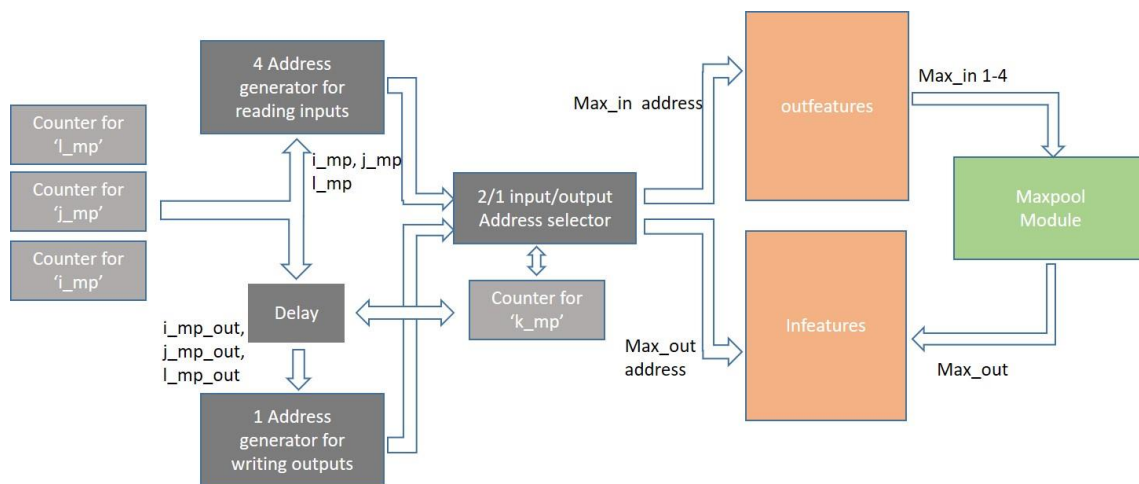


Figure 61 Block diagram for Maxpool layer

When Maxpooling is enabled, set of counters used for address index in Maxpool are incremented and addresses for input and output of Maxpool window are generated. 4 inputs are read from outfeatures where convolution results are written. These inputs are given to Maxpool module and result obtained is written back to Input feature memory again. The updated values in Infeatures memory are used for convolution in adjacent layer.

Single Maxpool module which picks one maximum value out of 4 is shown below.

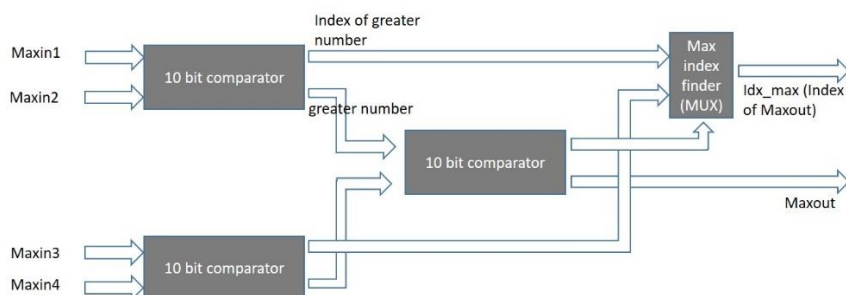


Figure 62 Maxpool module

It is implemented with 3 comparators which find greatest number. The module also gives the index of maximum value.

Device Utilization Summary (estimated values)				[-]
Logic Utilization	Used	Available	Utilization	
Number of Slices	42	4656	0%	
Number of 4 input LUTs	78	9312	0%	
Number of bonded IOBs	54	66	81%	
Number of GCLKs	1	24	4%	

Macro Statistics	
# Registers	: 12
Flip-Flops	: 12
# Comparators	: 6
10-bit comparator equal	: 3
10-bit comparator greater	: 3

Figure 63 Maxpool module resource utilisation

6.8 Max-finder Module

After final layer convolution is performed, 10 results are stored in registers and they are given to Maxpool module to find index of maximum value as shown below. Maxpool module is made to perform computation for 3 set of values to obtain maximum index of 10 values.

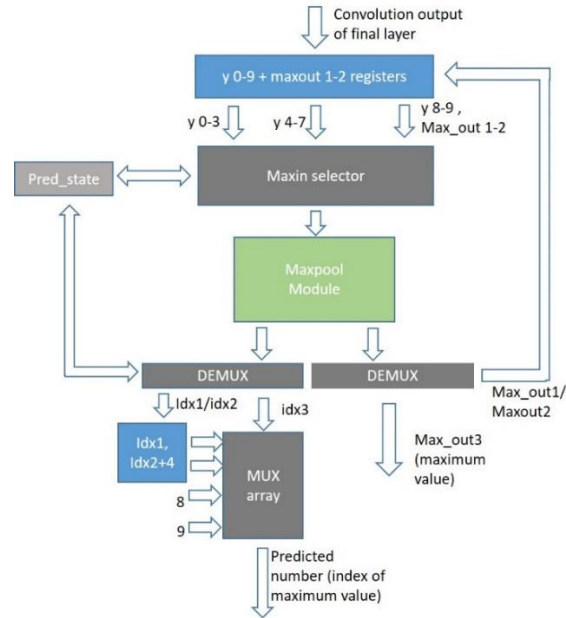


Figure 64 Block diagram for Max finder

In this way requirement of another module for max finder is avoided since index of maximum value is also generated to Maxpool module. The index of maximum after this operation is finished will correspond to the predicted digit/character.

6.9 Experimental Results

6.9.1 MNIST data set

Results of simulation when Input feature memory was given with an image file of digit 7 is shown below.

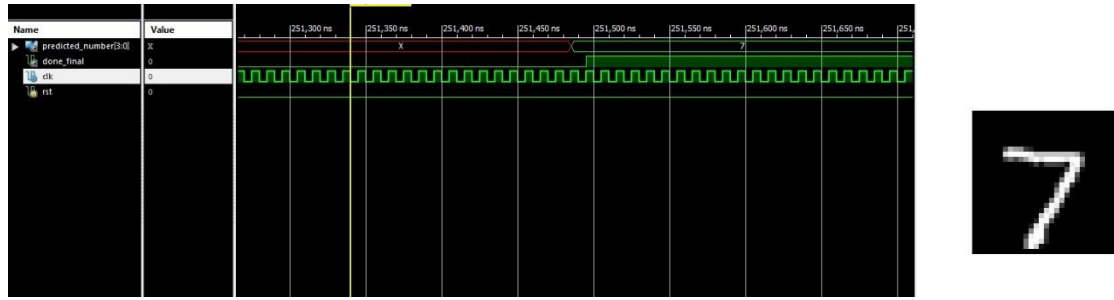


Figure 65 Prediction result for digit 7

Predicted number show the digit CNN architecture classifies. When prediction is done, it is indicated with a signal.

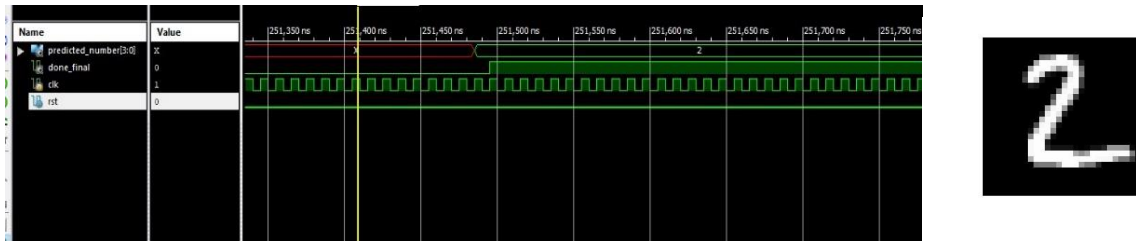


Figure 66 Prediction result for digit 2

Hardware utilization on spartan 3E FPGA after synthesis for with 2 different address generation circuits are shown below.

Device Utilization Summary (estimated values)				[-]
Logic Utilization	Used	Available	Utilization	
Number of Slices	2461	4656	52%	
Number of Slice Flip Flops	2062	9312	22%	
Number of 4 input LUTs	4452	9312	47%	
Number of bonded IOBs	7	66	10%	
Number of BRAMs	14	20	70%	
Number of MULT18X18SIOs	18	20	90%	
Number of GCLKs	1	24	4%	

Figure 67 Resource utilization with shifting in address generation.

```

Macro Statistics
# RAMs : 3
2790x10-bit dual-port RAM : 1
8000x10-bit dual-port RAM : 2
# ROMs : 2
64x10-bit ROM : 2
# Multipliers : 18
10x10-bit multiplier : 18
# Adders/Subtractors : 212
12-bit adder : 28
13-bit adder : 145
20-bit adder : 18
3-bit adder : 1
3-bit adder carry out : 1
5-bit adder : 5
5-bit adder carry out : 8
6-bit adder : 3
7-bit subtractor : 3
# Counters : 3
5-bit up counter : 3
# Registers : 262
1-bit register : 15
10-bit register : 173
12-bit register : 2
13-bit register : 4
2-bit register : 4
20-bit register : 38
4-bit register : 1
5-bit register : 25
# Comparators : 46
10-bit comparator equal : 3
10-bit comparator greater : 3
5-bit comparator equal : 6
5-bit comparator greater or equal : 5
5-bit comparator less : 7
5-bit comparator not equal : 6
6-bit comparator less : 4
7-bit comparator equal : 4
7-bit comparator greater or equal : 2
7-bit comparator less : 3
7-bit comparator not equal : 3

```

Figure 68 Hardware with shifting used in address generation

Device Utilization Summary (estimated values)				[-]
Logic Utilization	Used	Available	Utilization	
Number of Slices	2770	4656	59%	
Number of Slice Flip Flops	2068	9312	22%	
Number of 4 input LUTs	5064	9312	54%	
Number of bonded IOBs	7	66	10%	
Number of BRAMs	14	20	70%	
Number of MULT18X18SIOs	18	20	90%	
Number of GCLKs	1	24	4%	

Figure 69 Resource utilization with small multipliers used in address generation

```

Advanced HDL Synthesis Report

Macro Statistics
# RAMs : 5
2790x10-bit dual-port block RAM : 1
64x10-bit single-port block RAM : 2
8000x10-bit dual-port block RAM : 2
# Multipliers : 18
10x10-bit registered multiplier : 18
# Adders/Subtractors : 161
10-bit adder carry out : 1
12-bit adder : 32
13-bit adder : 92
20-bit adder : 18
3-bit adder : 1
3-bit adder carry out : 1
5-bit adder : 5
5-bit adder carry out : 5
6-bit adder : 3
7-bit subtractor : 3
# Counters : 3
5-bit up counter : 3
# Registers : 2521
Flip-Flops : 2521
# Comparators : 47
10-bit comparator equal : 3
10-bit comparator greater : 4
5-bit comparator equal : 6
5-bit comparator greater equal : 5
5-bit comparator less : 7
5-bit comparator not equal : 6
6-bit comparator less : 4
7-bit comparator equal : 4
7-bit comparator greater equal : 2
7-bit comparator less : 3
7-bit comparator not equal : 3
# Xors : 568
1-bit xor3 : 568

```

Figure 70 Hardware utilization with small multipliers used in address generation

It can be observed that total memory required is 14 BRAMs. This corresponds to 5 BRAMs for Infeatures and outfeatures with 8000 values and 3 for 2790 Weights and 1 for Bias storage. With use of shift instead of multiplication by constant number in the address generation of architecture LUT consumption is 7% less. But this has limited option to vary the number of channels as number of channels in each layer should be same with this hardware. Hence small multiplier modules are used in address generation part to accommodate any change in channel length.

6.9.2 Devanagari character task

Simulation results for Devanagari character classification showing which class the character belongs to out of 36 consonants is given below.





Figure 71 Prediction result for Devanagari dataset

As the number of channels is increased in each layer, number of weights also increases. This leads to change in storage space requirement. It also leads to changes in LUT logic utilisation as compared to MNIST data since number of bits required to represent address is changed. Resource utilization for character classification is shown in below figure.

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	3414	4656	73%	
Number of Slice Flip Flops	2772	9312	29%	
Number of 4 input LUTs	6150	9312	66%	
Number of bonded IOBs	10	66	15%	
Number of BRAMs	19	20	95%	
Number of MULT18X18SIOs	18	20	90%	
Number of GCLKs	1	24	4%	

```

Macro Statistics
# RAMs : 5
78x12-bit single-port block RAM : 2
8000x12-bit dual-port block RAM : 3
# Multipliers : 18
12x12-bit registered multiplier : 18
# Adders/Subtractors : 168
10-bit adder carry out : 1
13-bit adder : 124
24-bit adder : 18
3-bit adder : 1
4-bit adder : 2
5-bit adder : 6
5-bit adder carry out : 4
6-bit adder : 3
7-bit adder : 4
7-bit adder carry out : 1
7-bit subtractor : 2
9-bit subtractor : 2
# Counters : 3
5-bit up counter : 3
# Registers : 3324
Flip-Flops : 3324
# Comparators : 54
10-bit comparator greater : 1
12-bit comparator equal : 3
12-bit comparator greater : 5
5-bit comparator equal : 4
5-bit comparator greatequal : 2
5-bit comparator less : 4
5-bit comparator not equal : 4
6-bit comparator less : 4
7-bit comparator equal : 8
7-bit comparator greatequal : 5
7-bit comparator less : 5
7-bit comparator not equal : 5
9-bit comparator equal : 2
9-bit comparator less : 1
9-bit comparator not equal : 1
# Xors : 568
1-bit xor3 : 568

```

Figure 72 Resource utilization for Devanagari character recognition using 12 bit

12-bit quantization is used in Devanagari character classification due to considerable accuracy drop in 10-bit. With change in data quantization also there is a change memory utilization along with some extra LUT resource utilisation.


Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	3092	4656	66%	
Number of Slice Flip Flops	2363	9312	25%	
Number of 4 input LUTs	5667	9312	60%	
Number of bonded IOBs	10	66	15%	
Number of BRAMs	16	20	80%	
Number of MULT18X18SIOs	18	20	90%	
Number of GCLKs	1	24	4%	

Figure 73 Resource utilization of Devanagari recognition with 10-bit quantization

From two figures, it can be observed that number of Slices increased 7% and BRAM requirement increased 15% by changing from 10-bit to 12 bits. As with 12 bits BRAM is almost used, datawidth can't be increased further with this architecture. So, 12-bit quantization gives better accuracy with maximum effective utilization of resources.

6.10 Summary

Hardware architecture details for inference of handwritten digit and character recognition tasks were discussed. After obtaining suitable CNN architecture with good accuracy and limited memory requirement, the way it can be transferred to hardware implementation was explored in this chapter. The goal was to make use of parallelism and reduce memory access in computation. Fast and flexible storage with BRAM was effectively utilized. The limited resources availability of low-end FPGAs was taken care while selecting convolutional networks. We have seen that systolic architectures can be very useful in hardware implementation of convolution since cost of reading and writing in external memory plays key factor in convolution operations. Basic idea of what systolic architectures intends to do and levels of parallelisms possible in CNN were discussed. Block RAMs in FPGA were discussed briefly to get basic idea of how it can be used as memory element while implementing hardware. We have seen different layers of CNN i.e., convolution, Pooling and Activations can be implemented as hardware and how computational blocks are connected to memory blocks. Design aspects in dealing with Fixed-point data used in FPGA were also mentioned.

7 CONCLUSION AND FUTURE WORK

7.1 Contribution of the Thesis

In this project, a CNN architecture for digit recognition on MNIST dataset and Devanagari Character recognition for a low end FPGA device were presented. Strategic approaches to obtain optimal performance and accuracy were demonstrated along with experimental results. Quantization strategy for fixed data points were also presented with experimental results. Then a VLSI architecture and implementation of character recognition on Low end FPGA device were presented. Design flow of different components along with experimental results were presented. A study on CNN algorithms and optimization with numerical examples were also carried out.

In third chapter, fundamental approaches in machine learning and optimization algorithms were discussed. A multi-layer perceptron model with mathematical details of the gradient computations were examined. In fourth chapter, Advantages of inclusion of convolutional layers into deep learning networks and their computational details were examined with numerical example. The effects of Maxpooling, strides and padding in convolution were also discussed.

In fifth chapter, Training the CNN algorithms for MNIST and Devanagari datasets for improved performance by taking hardware design constraints into consideration were presented with experimental details. Quantization strategy to reduce accuracy loss while dealing with fixed data points were examined with experimental results.

In sixth chapter, a VLSI architecture for digit and character recognition tasks using systolic design for convolution were presented. Design process of Convolution, Maxpooling and RELU components along with control blocks were discussed. The experimental results of implemented architecture along with design process of different components such as convolutions, Maxpooling and RELU were presented.

7.2 Future Work and Extension

Architecture should be made reconfigurable to accommodate different filter sizes other than 3x3 filters. Scalability of the architecture to adapt into different devices with different memory and multiplier resources should be explored. Interfacing the implemented design on FPGA with image capturing devices, external storage devices etc. to deploy in real life applications has to

be carried out. Scope of different activation functions which can reduce accuracy drop in quantization without need of any quantization strategy can be explored. Scope of any customized activation functions with less computational complexity to use in final layer in place of computationally/memory expensive SoftMax which can give approximate prediction probabilities along with predicted class can also be explored. The same architecture can be employed to do both forward and backward propagation to train the deep network using FPGAs.

REFERENCE

- [1]. Ian Goodfellow, Deep learning, <https://www.deeplearningbook.org/>.
- [2]. Glorot, X., Bordes, A. & Bengio. Y,” Deep sparse rectifier neural networks”, In Proc. 14th International Conference on Artificial Intelligence and Statistics 315–323 (2011).
- [3]. Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner, “Gradient Based learning applied to Document recognition”, in proc of the IEEE, 1998.
- [4]. M.D Zeiler and R.Fergus, “Visualizing and understanding convolutional networks,” in Proc. Eur. Conf. Comput. Vis. Zurich, Switzerland, 2014.
- [5]. A.Krizhevsky, I.Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in NIPS, 2012, pp. 1097–1105.
- [6]. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” arXiv preprint arXiv:1409.1556, 2014.
- [7]. M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” 2016.
- [8]. Matthieu Courbariaux, Yoshua Bengio, Jean-Pierre David, “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”, arXiv:1511.00363v3 [cs.LG] 18 Apr 2016.
- [9]. S. Acharya, A. K. Pant and P. K. Gyawali, Deep learning based large scale handwritten devanagari character recognition, in *2015 9th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, (Dec 2015), pp. 1–6.
- [10]. S. Arora, D. Bhattacharjee, M. Nasipuri, D. K. Basu and M. Kundu, Combining multiple feature extraction techniques for handwritten Devnagari character recognition, in *2008 IEEE Region 10 and the Third Int. Con. Industrial and Information Systems (2008)*, pp. 1–6.
- [11]. Aarati Mohite, Sushama Shelke, “Handwritten Devanagari Character Recognition using Convolutional Neural Network”, in 2018 4th International Conference for Convergence in Technology (I2CT).
- [12]. R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor and S. Areibi, "Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks," *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 265-268, doi: 10.1109/FPT.2016.7929549.

- [13]. C. Zhang, et al., "Optimizing FPGA-based accelerator design for Deep Convolutional Neural Networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161-170, February 2015.
- [14]. M. Motamedi, P. Gysel, V. Akella and S. Ghiasi, "Design space exploration of FPGA-based Deep Convolutional Neural Networks," *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 575-580, doi: 10.1109/ASPDAC.2016.7428073.
- [15]. K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.
- [16]. Yufei Ma, N. Suda, Yu Cao, J. Seo and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1-8.
- [17]. Y. Chen, J. Emer and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367-379.
- [18]. J. Jo, S. Kim and I. Park, "Energy-Efficient Convolution Architecture Based on Rescheduled Dataflow," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4196-4207, Dec. 2018.
- [19]. W. Lu, G. Yan, J. Li, S. Gong, Y. Han and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 553-564.
- [20]. Kung H. T. and W. Song. "A Systolic 2-D Convolution Chip, Hb".
- [21]. Kung, H. T.. "Why systolic architectures?" *Computer* 15 (1982): 37-46.
- [22]. Y. Parmar and K. Sridharan, "A Resource-Efficient Multiplierless Systolic Array Architecture for Convolutions in Deep Networks," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 2, pp. 370-374, Feb. 2020.
- [23]. J. Qiu, et al., "Going deeper with embedded FPGA platform for Convolutional Neural Network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 26-35, February 2016.
- [24]. M. Cho and Y. Kim, "Implementation of Data-optimized FPGA-based Accelerator for Convolutional Neural Network," *2020 International Conference on Electronics, Information, and Communication (ICEIC)*, 2020, pp. 1-2.