CNN on ASIC

Handwritten digit recogniser

MID-SEM EVALUATION REPORT

Project Guide

Dr. Ramesh Kini

Department of Electronics and Communication Engineering

National Institute of Technology Karnataka, Surathkal

Group Members

Achyut Shegade - 191EE101

Rahul Royal - 191EE208

Govardhan L. R. - 191EE215

Madhur Sharma - 191EE132

EC383 - Mini Project in VLSI Design

Jan 2022 - April 2022

Introduction

A typical multi-layer neural network is convolutional neural networks (CNNs). Deep technologies are now widely applied in the disciplines of machine vision and voice analysis, thanks to the ongoing development of deep technology. Convolutional neural networks (CNNs), a type of artificial neural network that has been popular in computer vision, are gaining popularity in a variety of fields, including radiology.

Traditional convolutional neural networks use CPUs to carry out calculations. Such calculations are wasteful and slow, and thus make it impossible to meet real-time calculating needs. As a result, convolutional neural networks based on graphics processing units (GPUs) are commonly used. The literature evaluated CNN open source projects that use GPUs and discovered that GPUs have some drawbacks, such as high power consumption and high cost. The major disadvantage of CNN is that it is a computationally costly model. Taking this into account, FPGA is a better fit as it is a low power device, which provides a similar degree of acceleration to a GPU.

A field-programmable gate array (FPGA) is a programming hardware circuit structure that can be customized and is a common digital circuit design approach. The parallel computing mode provided by FPGA is compatible with the computational properties of convolutional neural networks. At the same time, the reprogrammable features of FPGA make it ideal for neural networks with a changing network structure. As a result, the design of a CNN based on FPGA has gotten a lot of interest. The convolutional neural network uses several convolution layers, pooling layers, and fully connected layers to produce the recognition result from the image that will be identified as the input. The CNN's main building block is the convolution layer. It is responsible for the majority of the network's computational load.

The aim of this project is to create an efficient design for Convolutional Neural Network Algorithm to match the required computational power and speed that is difficult to be obtained using general purpose processors. This design can be used in real time Computer Vision Applications.

Literature Survey

The research on the artificial neural networking and convolutional neural networking in recent days is gaining huge scope because of racing artificial intelligence(AI) in every field. The interest is to mainly address image recognition and to get data into forms that are easier to process without losing the features that are important for figuring out what the data represents and to increase the speed and reduce the energy consumption of the learning process of neural networks by performing forward and backpropogation in hardware. We were successful in completing the code for different layers of CNN and link all the modules in a control module with in time. Construction solution for implementation memory neural network using FPGA is described. In every case, the performance of the implemented system is lower when the number of associations are increased. Our future goals are to increase size of the images and implement other learning algorithms for better performance of neural network. In this section, we explore related survey articles published over the years

discussing CNN implementation on FPGA. Some surveys covered neural networks implementations on hardware in general . Other surveys focused on FPGA-based accelerators for deep learning neural network .In [Adiono et al., 2018], the authors focused on the FPGA implementation of convolutional neural networks (CNNs). In [Yogatama et al., 2018],the author discussed about learning algorithm architecture and recognizing algorithm architecture. In [**yogatama2018fpga**] the author, mentioned things about hardware architecture and about the several methods in implementing and we utilize McLaurin series to approximate the sigmoid function. To the best of our knowledge, our project is the most recent survey with comprehensive coverage discussing the recent work in the area of implementing CNN on FPGAs, GPUs and ASIC. This work complements the existing work and contributes towards providing the complete background on CNN. The contributions of this survey can be summarized as follows:

- This project consist of systematic Verilog coding (Xilinx vivado) for different layers of CNN.
- We used Tensorflow to develop and train models using python because it is not easy to train on hardware directly as the training algorithms require a lot of hardware.
- The weights and biases obtained from after training are stored directly into block RAMs.
- Testbenches have been written for all modules to test the functional modules individually

The table below shows the comparison between GPUs, FPGAs and ASICs and it is notable that FPGA and ASIC designs can provide various benefits over the others.

Metric	GPUs	FPGAs	ASICs
Frequency	High	Low	Medium
Power	High	Medium	Low
Energy Efficiency	Low	Medium	High
Area	Large	Large	Small
Performance	Low	Medium	High
Unit Cost	Medium	High	Low

Table 1: Comparison of GPUs, FPGAs and ASICs Moolchandani, A. Kumar, and Sarangi, 2021

A 28*28 image of a hand written digit is taken as the input and the motive is to detect this digit at the output. In CNN algorithms, we have to preprocess the image using convolution and maxpooling and feed these processed values to a fully connected layer with softmax as the output layer. This pre-processing reduces the number of hidden layers required and also increases the accuracy of the model. We can tune the model by increasing the number of convolution layers, increasing the number of filters, using optimal number of neurons required. We have also tried to optimize the hardware requirement by using concepts like pipelining.
We also explored about the pooling options that are available. In a convolutional neural

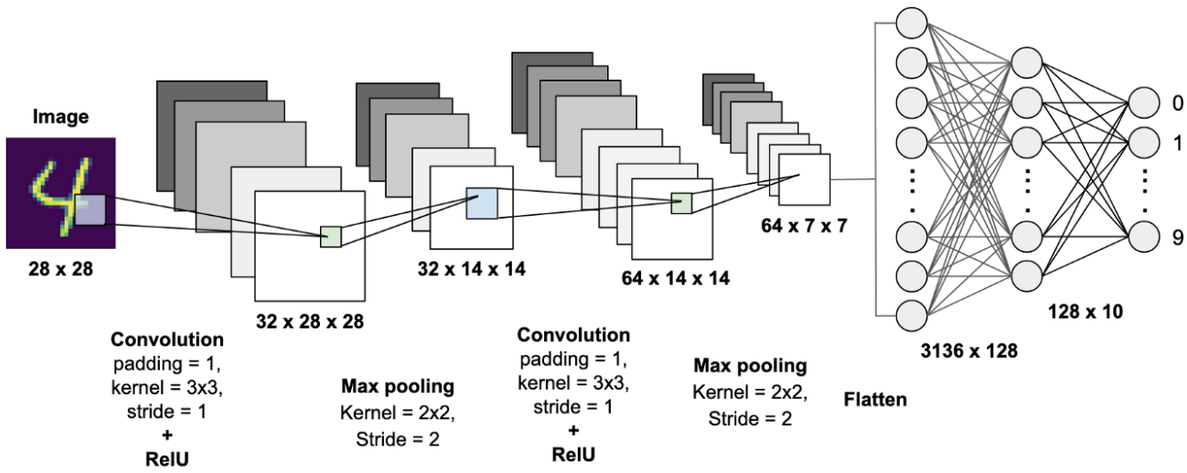

Figure 1: Layers in CNN

network, pooling layers are applied after the convolutional layer. The main purpose of pooling is to reduce the size of feature maps, which in turn makes computation faster because the number of training parameters is reduced. The pooling operation summarizes the features present in a region, the size of which is determined by the pooling filter. If a filter has the dimensions of 2x2, then the region that is summarized is also of the size 2x2.

Types of Pooling Operations

- Max Pooling
- Min Pooling
- Average Pooling
- Global Pooling

Max Pooling

In this type of pooling, the summary of the features in a region is represented by the maximum value in that region. It is mostly used when the image has a dark background since max pooling will select brighter pixels.

Min Pooling

In this type of pooling, the summary of the features in a region is represented by the minimum value in that region. It is mostly used when the image has a light background since min pooling will select darker pixels.

Average Pooling

In the third type of pooling, the summary of the features in a region are represented by the average value of that region. Average pooling smooths the harsh edges of a picture and is used when such edges are not important.

Global Pooling

Each channel in the feature map is reduced to just one value. The value depends on the

type of global pooling, which can be any one of the following:

Global Max Pooling, Global Min Pooling and Global Average Pooling
Global pooling is almost like applying a filter of the exact dimensions of the feature map.

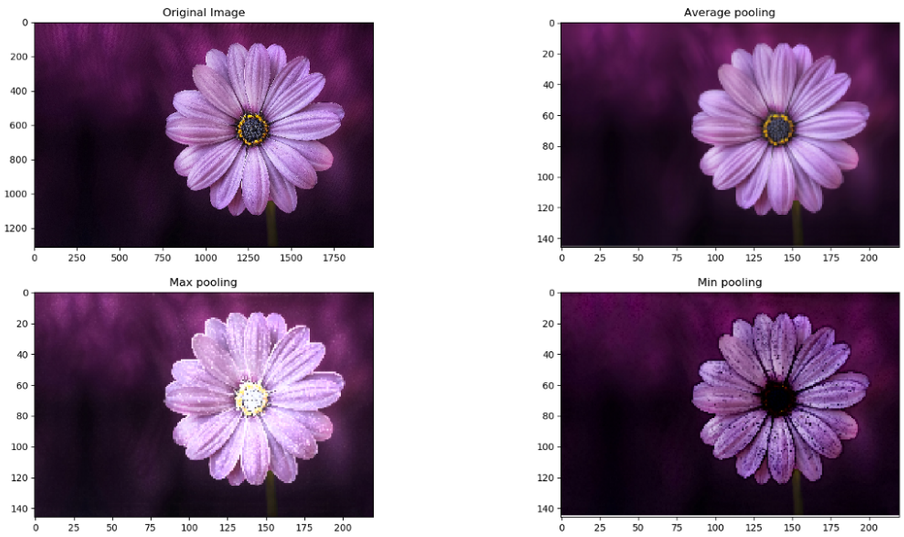

Figure 2: Different Pooling Operations on an image

Benefits of Pooling

- Faster computation due to reduced parameters
- Robust to variations in the features' positions such that if the feature exists in a different position than it did in the training data, it can still be accurately classified
- Reduce over-fitting

Drawbacks of Pooling

- Some spatial information may be lost, especially if the filter size is too big
- If max pooling is used in an image where the background is light, it may eliminate the foreground features and only return the background (since white has a pixel value of 255 and black 0). Same goes for dark background and min pooling.

Outputs Obtained after Pooling

The size of the feature map after the pooling layer is:
((l - f + 1) / s) * ((w - f + 1) / s) * c
Here, l = length of the feature map
w = width of the feature map
f = dimensions of the filter
c = number of channels of the feature map
s = stride

Project and Implementation Details

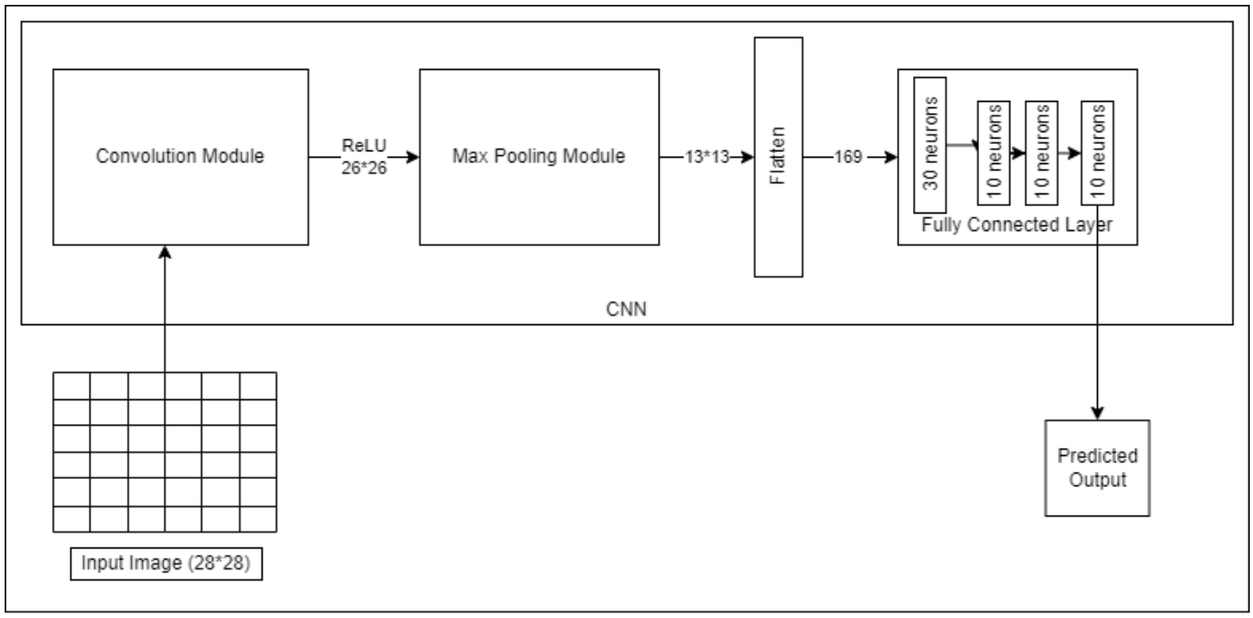

Figure 3: Overall Architecture

Convolution layer

In the convolution layer, the image taken in terms of a matrix is convolved with a kernel matrix which can detect features from image based on the values it stored in it. The image and the kernel size used in this module are parametrized and can be changed if needed but for this project, the image size of 28*28 and kernel size of 3*3 is considered.
The various blocks used to implement this convolution are:

- Memory Unit(RAM):
  Image is considered to be a 28*28 matrix having 784 elements each of which can take values ranging from 0 to 255 where 0 and 255 represent black and white respectively. The image is stored in this memory unit in terms of these 784 elements (0 to 783). Kernel is considered to be a 3*3 matrix having 9 elements each assigned a value depending on the feature we wish to detect. Here we are using 64 different filters. Hence we need 64*3*3 values to be stored in memory.

- Image Address Generator(IAG):
  This block generates addresses of required pixels for convolution of image that is from 0 to 783 which are used to access their element values from the memory.

- Kernel Address Generator(KAG):
  This block generates addresses of kernel in a serial manner which are used to get the kernel element values from memory.

- Computational block:
  This block perform computation that is it multiplies the kernel values with the individual element values from the image and sends the value to the storage block.

- Control block:
  This block basically controls the entire convolution operation from reading the image values from memory to storing back the result after the process.

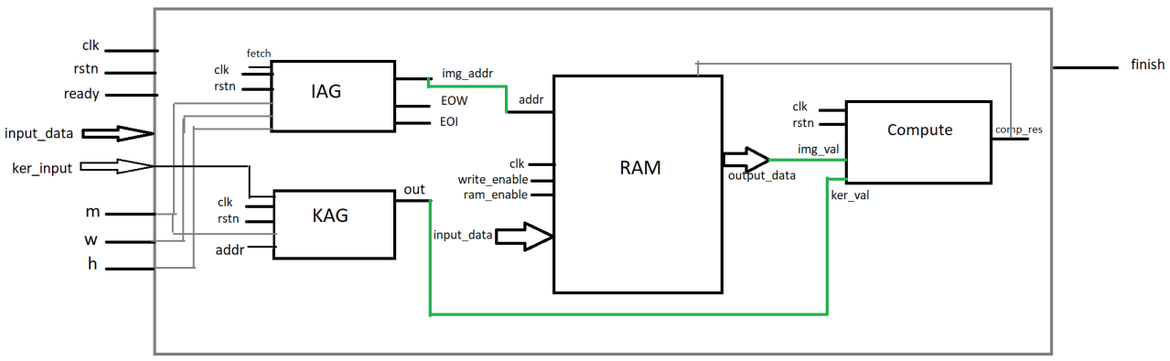

Figure 4: Block Diagram of Convolution Module

The block diagram and simulation results of convolution module can be found below,

Max Pooling layer

This layer is used to reduce the size of the image extracting the dominating features of the image and neglecting others ensuring that overall structure of the image is still maintained. In this project, MaxPooling is done with a kernel size of 2*2, that is if the input image size is M*N then the output from this layer will be of size M/2*N/2.

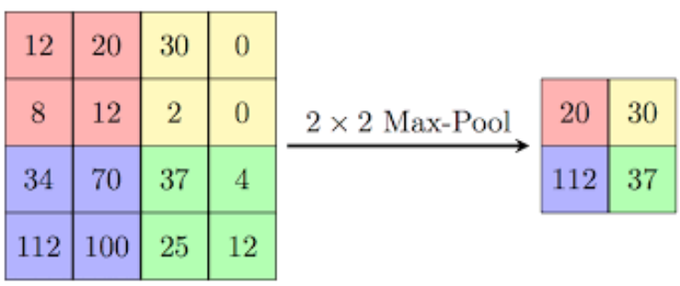

Figure 5: Max Pooling with 2*2 kernel with a stride of 2

The various blocks used to implement this convolution are:

- Memory Unit:
  This is used to store the image obtained after the convolution and ReLU operations.
- Image address generator:
  This block generates addresses of image with which we can access their respective values from the memory unit.
- Computational block:
  This block perform computation that is it takes values of four elements from the feature image, two from one row and other two from immediate next row and finds the maximum among them and sends the result to the storage block.
- Storage block:
  This block takes the result from the compute block and stores it. After the entire process, the maxpooled output will have an image size of N/2 *N/2.

- Control block:
  This block basically controls the entire maxpooling operation from reading the outputs of previous convolution layers, performing maxpool and storing the result.

  The block diagram of max pool module can be found below (Figure 5)

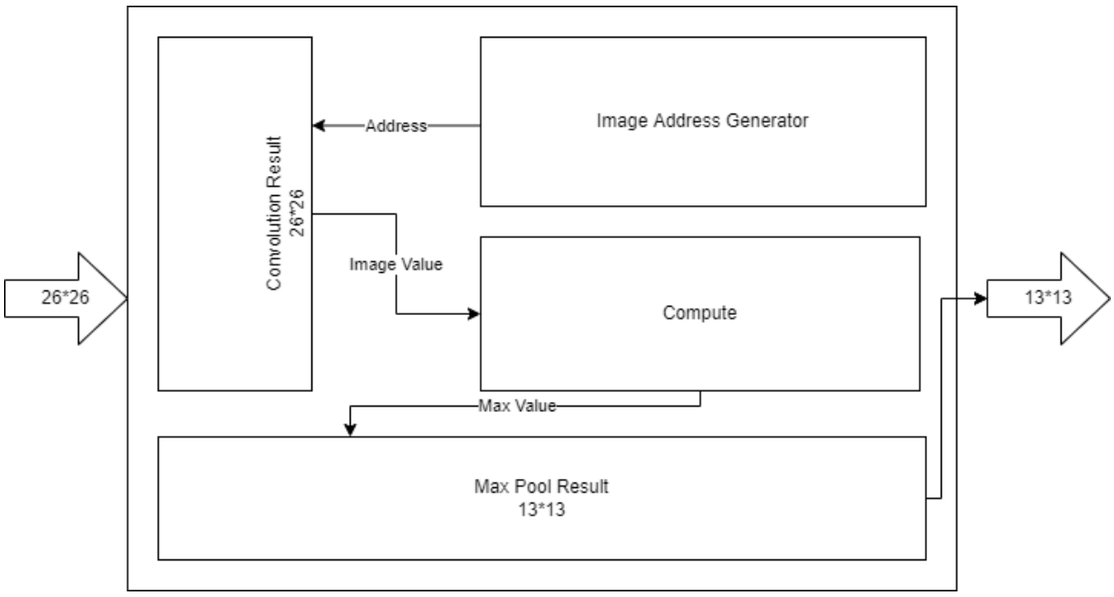

Figure 6: Max Pool Block Diagram

ReLU Activation layer
In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.

The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

Forward Neural Network
Fully Connected Neural Network composed of multiple layer and each layer have a fixed number of neurons. All these layers also include input layer and output layer and remaining are the hidden layers.
Since the Neural Network is built by connecting many neurons in a particular fashion so it is necessary to discuss the general structure of Neuron and how it is implemented on Hardware.
Neuron
A neuron have many inputs and one output. Each input is multiplied by a corresponding weight and an overall bias is added. After that this result is applied to an activation function that gives the finial output.

Neuron is implemented in Verilog in the following way:

Note that every neuron will be accepting the inputs sequentially with positive edge of clock. In a neuron two external modules are instantiated both of them are described below:

- Weight Memory:
  Since we are using pretrained neural network so we will store all the weights for a particular neuron in the weight memory. Here weight memory is implement as ROM.

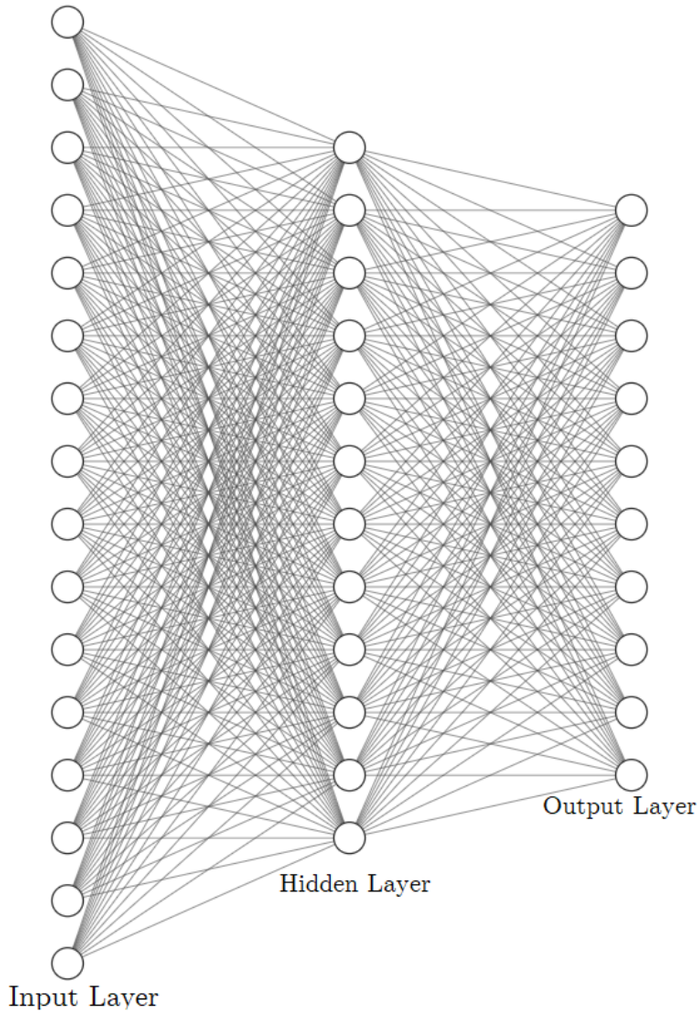

Figure 7: Fully Connected FNN

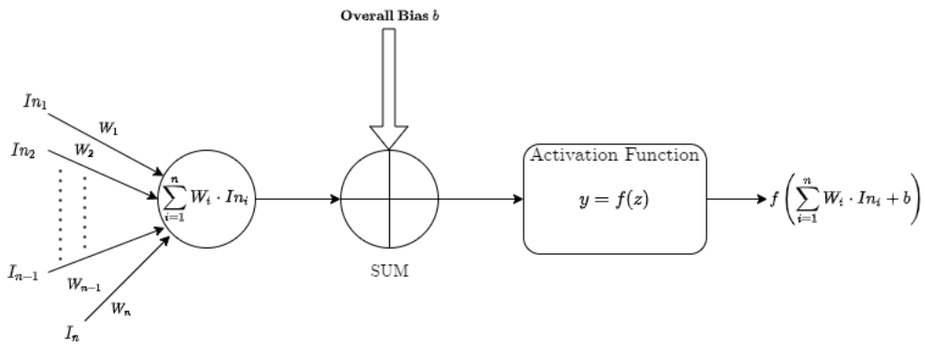

Figure 8: General Neuron

Number of weights will be equal to the number of inputs and this will also decide the depth of memory.

- Activation Function:
  After finding $b + \sum w_i I_i$ this will be applied to an Rectified Linear activation function which will given the finial output. Note that here b is overall bias, w_i represent the i_{th} weight stored in weight memory and I_i is the i_{th} input.

The process of finding $b + \sum w_i I_i$ can be described as follow:
Input will be coming sequentially one at a time. At every clock edge we will read the corresponding weight from the weight memory and multiply it with the incoming input.

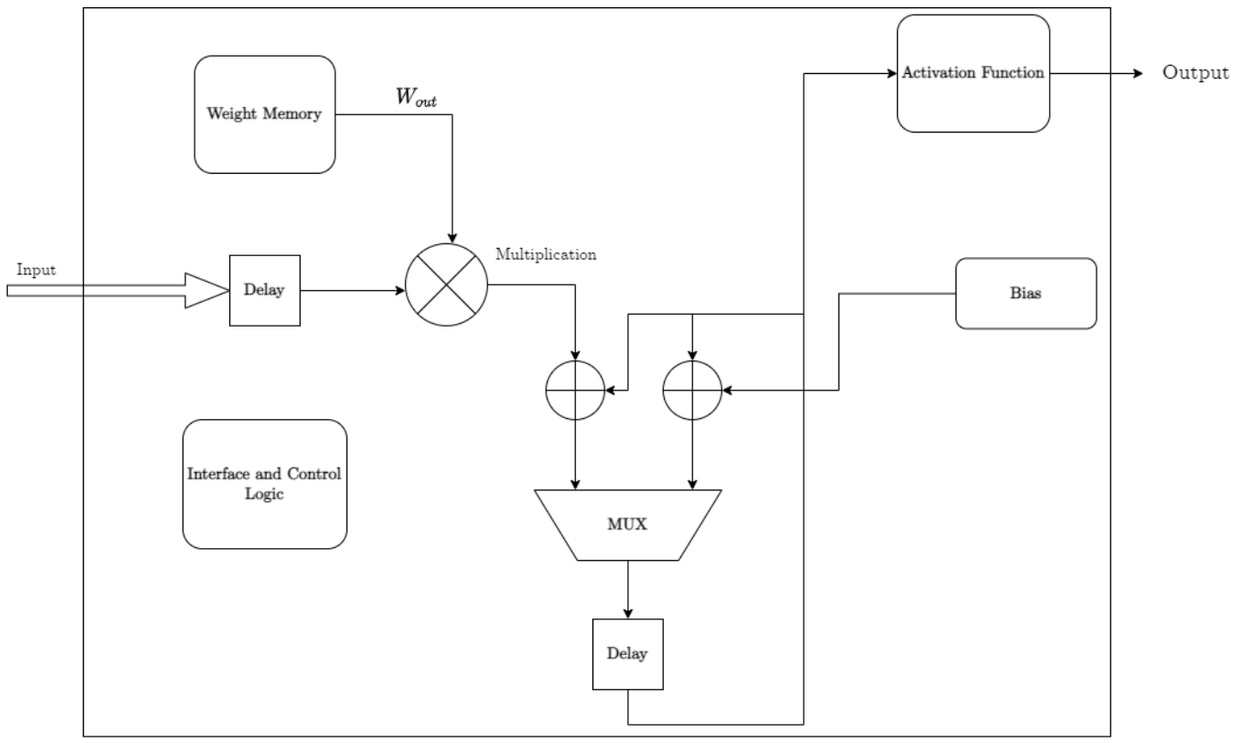

Figure 9: Implementation of Neuron

After that this will be stored in a register. Now, the next input will come, so the next weight will be read and multiply with it. This new multiplication result will be added to the previously value stored in the register. This process will continue untill we reach the last address of the memory. After reaching to the last address this sum will be added to overall bias b. In this way we will get $b + \sum w_i I_i$.
After this, overall sum will be sent to rectified linear activation function module which will calculate the finial output of neuron.

Design of Neural Layers and their Connection:
Another important thing in the design of fully connected FNN is design of layer and connection of neurons between two consecutive layers. This is described below:

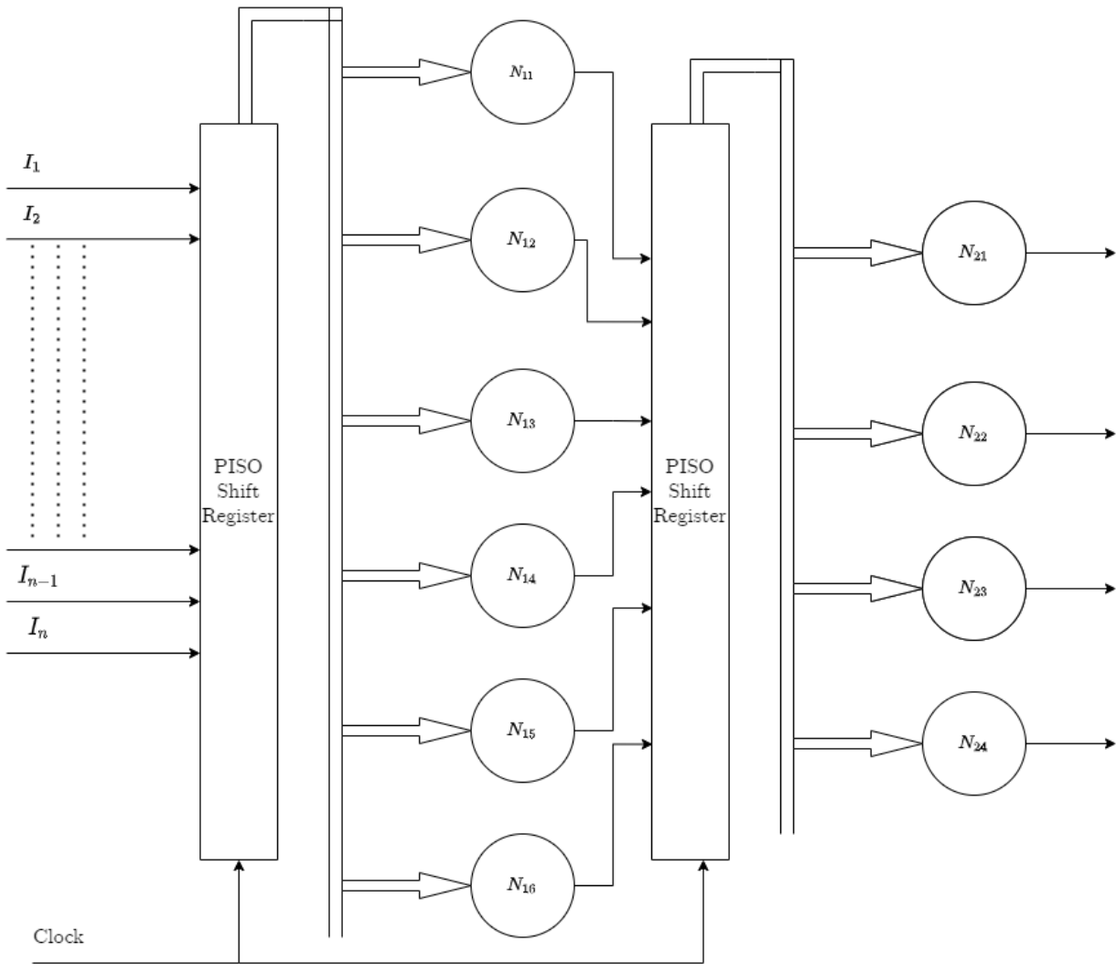

Figure 10: Neural Layers Design in Verilog and connection of neurons or FFN

Since our every neuron accept the input one at a time so first of all the coming inputs needs to be converted from parallel to serial format. This is implemented by first storing all the coming input in a $(N \times W)$ width register and then output is taken at every clock edge by shifting this stored value by W bits. Note that here N is the number of inputs and W is data width. Same thing will be done between first and second layer and so on.

Now the output of this PISO shift register will be fed to each neuron of next layer and after that next layer neurons will calculate corresponding output according to their functionality. Similarly output of finial layer will be converted to serial format and this will be fed to next module called soft max which is described next.

Soft Max:

- General Formula:

$$\sigma(\overline{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Softmax-Layer.v:
  This module intakes the 1D array and returns the max value out of it. It first instantiates **exponential.v** to get the exponential array and then applies **Accumulator.v**, to

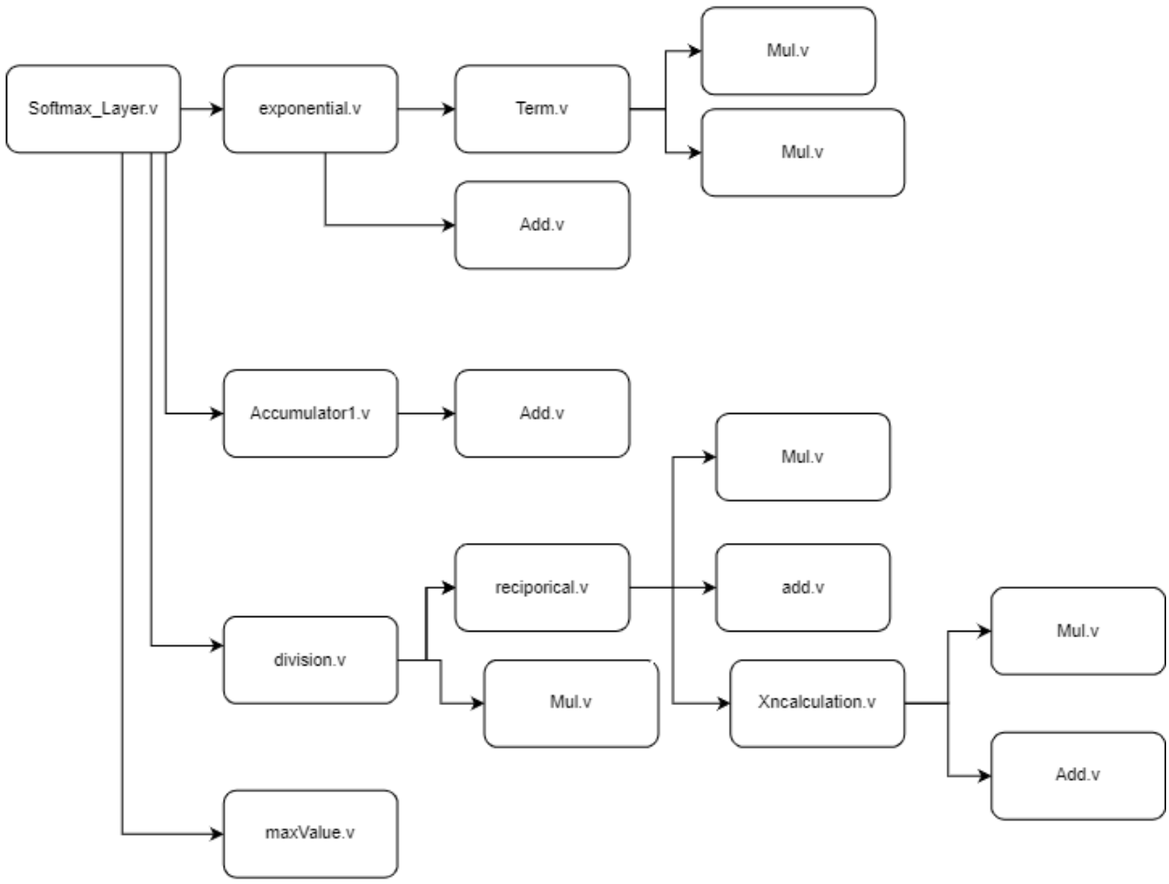

Figure 11: Soft Max Module

get the summation of the exponential array and further applies **division.v**, to calculate the probability of the elements in the array and then finally applies **maxValue.v** to find out the maximum out of the array.

- Exponential.v:
  This module is to calculate exponential pf floating point input by using Taylor expansion method. It calls module **term.v** that multiply input number by its powered then multiply by constant by using **mul.v** modules and calls addition to sum the calculated 6 terms.

- Accumulator1.v:
  This module calculates the total by adding all the elements in the exponential array.

- Division.v:
  This module calls in the reciprocal module which uses Newton – Raphson method to find the reciprocal of the divisor and then multiplies with the reciprocal with the X_n which is obtained by the module X_n Calculation.

- maxValue.v:
  This module loops on to get the maximum by comparing two consecutive numbers' exponent in array to get the max, if two exponents are equal, then it compares the mantissas of the numbers.

Results

An optimal architecture was developed with least number of parameters while maintaining the accuracy as high as possible. The total number of trainable parameters was reduced to 34,370 which was above 6,00,000 if we had not used the convolution and max pooling layers. The number of parameters in each layer can be found below

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 1)	10
activation_5 (Activation)	(None, 26, 26, 1)	0
max_pooling2d_1 (MaxPooling 2D)	(None, 13, 13, 1)	0
flatten_1 (Flatten)	(None, 169)	0
dense_4 (Dense)	(None, 169)	28730
activation_6 (Activation)	(None, 169)	0
dense_5 (Dense)	(None, 30)	5100
activation_7 (Activation)	(None, 30)	0
dense_6 (Dense)	(None, 10)	310
activation_8 (Activation)	(None, 10)	0
dense_7 (Dense)	(None, 10)	110
activation_9 (Activation)	(None, 10)	0
dense_8 (Dense)	(None, 10)	110
activation_10 (Activation)	(None, 10)	0

Total params: 34,370

Figure 12: Number of Parameters of different layers

```
[21] model.fit(x_trainr,y_train,epochs=5,validation_split=0.3)

Epoch 1/5
1313/1313 [==============================] - 16s 12ms/step - loss: 0.6067 - accuracy: 0.8021 - val_loss: 0.2926 - val_accuracy: 0.9131
Epoch 2/5
1313/1313 [==============================] - 14s 11ms/step - loss: 0.2145 - accuracy: 0.9374 - val_loss: 0.2108 - val_accuracy: 0.9391
Epoch 3/5
1313/1313 [==============================] - 14s 11ms/step - loss: 0.1596 - accuracy: 0.9523 - val_loss: 0.1908 - val_accuracy: 0.9443
Epoch 4/5
1313/1313 [==============================] - 14s 11ms/step - loss: 0.1283 - accuracy: 0.9616 - val_loss: 0.1477 - val_accuracy: 0.9581
Epoch 5/5
1313/1313 [==============================] - 14s 11ms/step - loss: 0.1109 - accuracy: 0.9664 - val_loss: 0.1452 - val_accuracy: 0.9587
<keras.callbacks.History at 0x7fdfbde90c50>

test_loss,test_accuracy = model.evaluate(x_testr,y_test)
print(test_loss)
print(test_accuracy)

313/313 [==============================] - 2s 5ms/step - loss: 0.1301 - accuracy: 0.9639
0.13008606433868408
0.9639000296592712
```

Figure 13: Accuracy

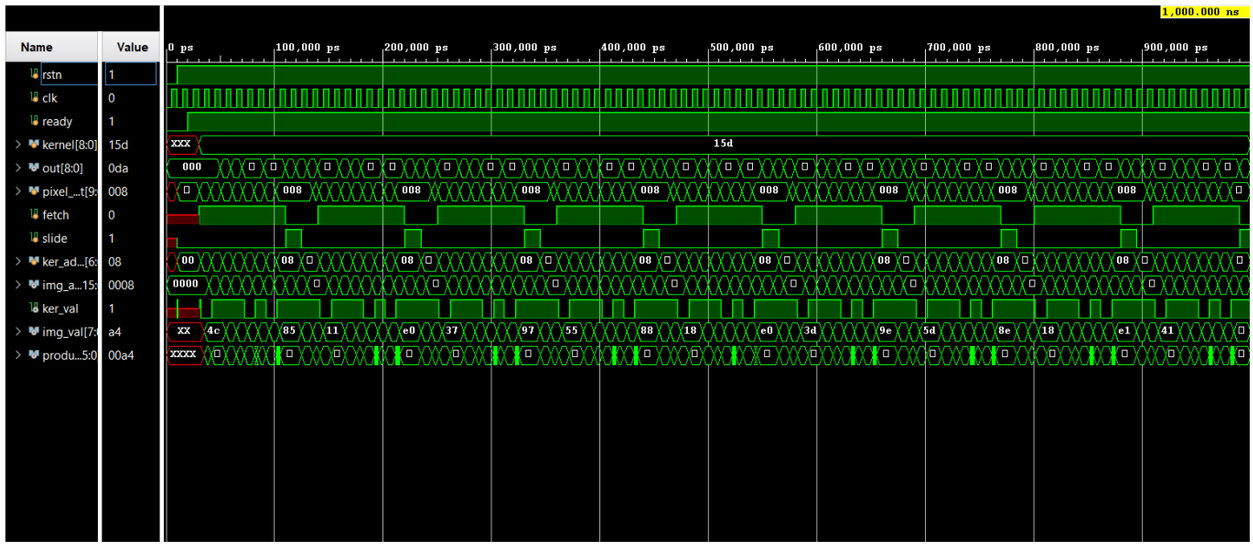

Figure 14: Simulation result of Convolution Module

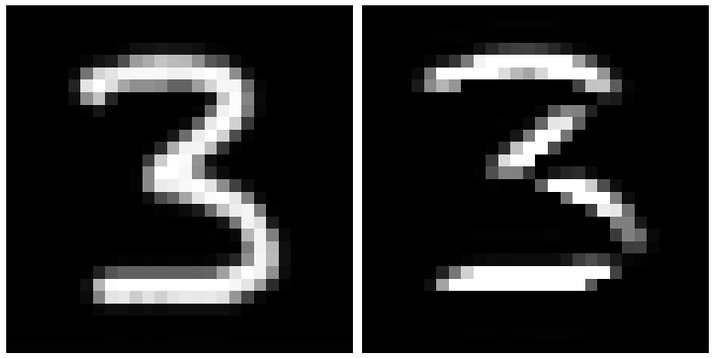

Figure 15: Result of convolution Module

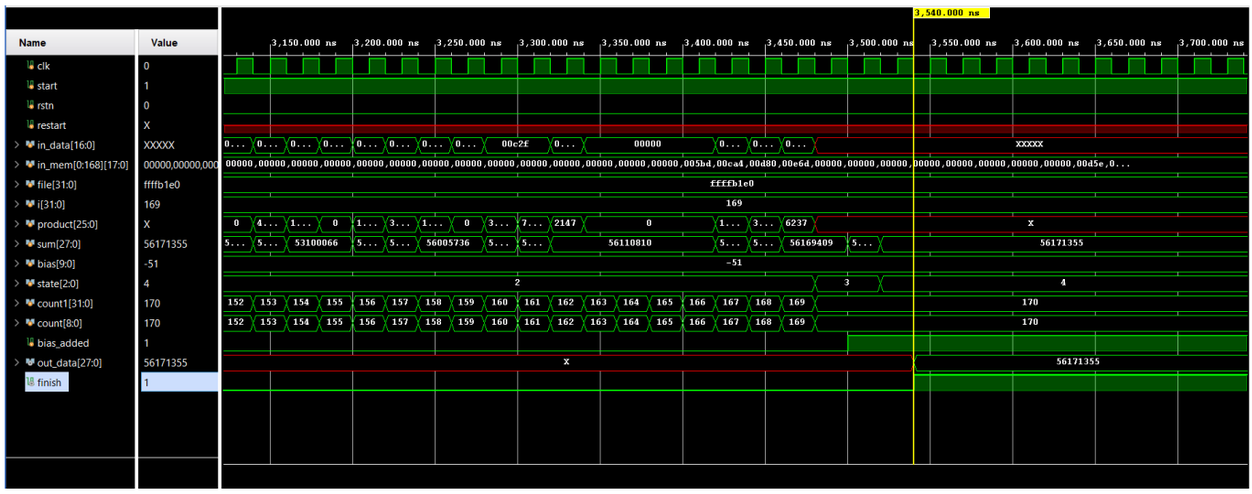

Figure 16: Simulation Result of Neuron Module

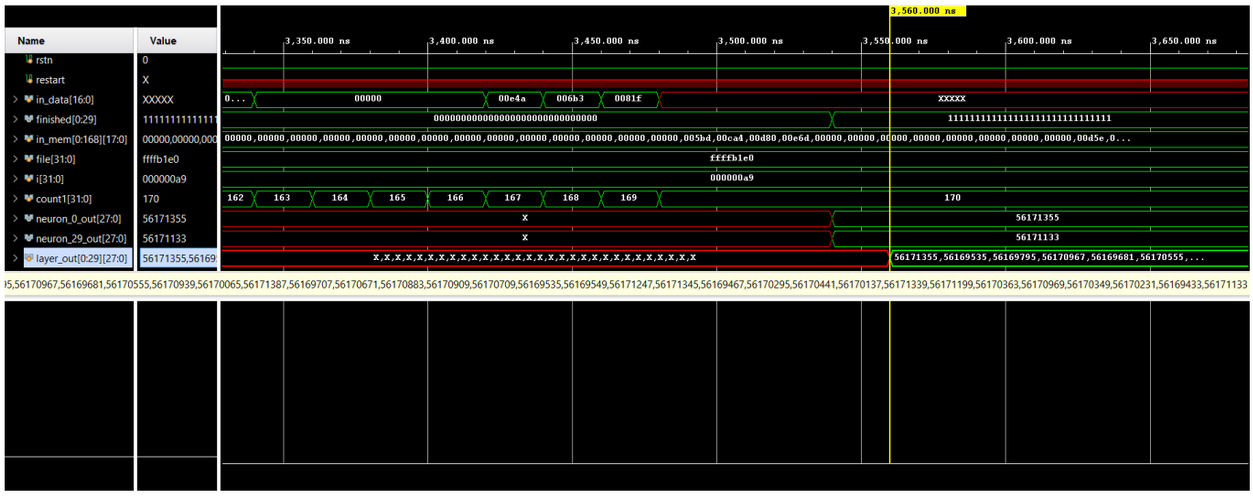

Figure 17: Simulation Result of Layer Module

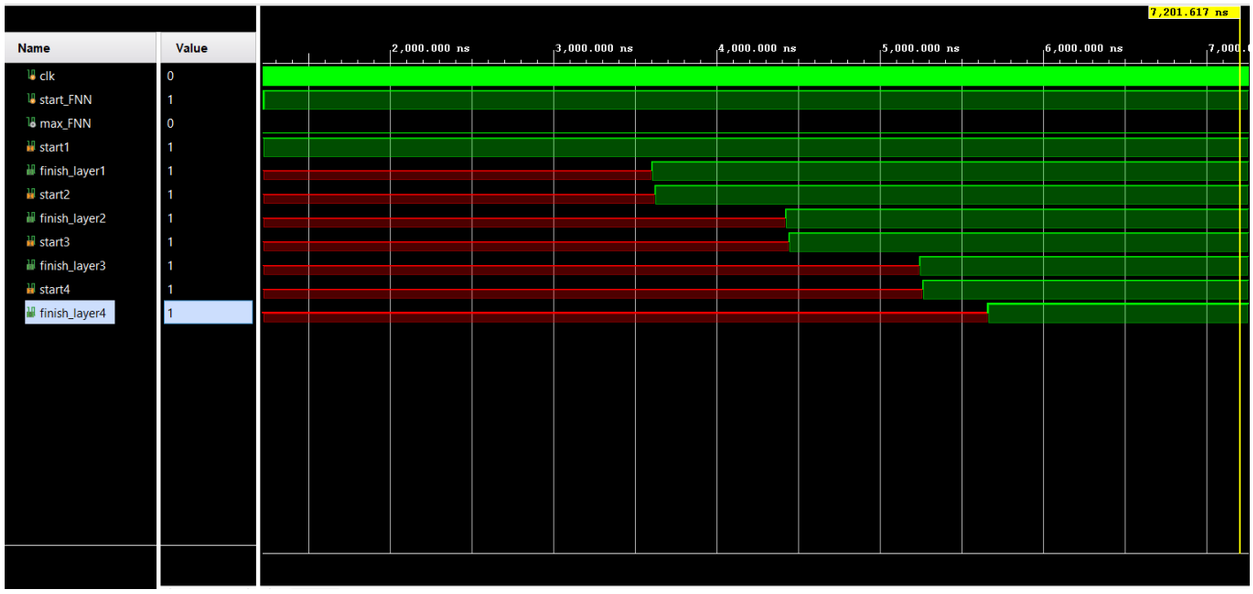

Figure 18: Simulation Result of FNN Module

Figure 19: Final result simulation

Conclusion

A hand written digit recogniser is designed and can be implemented on FPGA. This design can be further optimized to improve the performance which can then be used in real time applications.

Appendix

0.1 Code Repositories

The codes of all the modules can be found in the following github repository `https://github.com/achyutshegade/CNN-on-ASIC.git`

0.2 References

All the references used can be found on the next page:
Yogatama et al., 2018 Jing, 2016 Berchmans and S. Kumar, 2014 Adiono et al., 2018 Xiao, Shi, and Zhang, 2020 Leiner et al., 2008 Adiono et al., 2018 **amjad˙value˙2020 cohen˙quantifying˙2020 hawking˙particle˙1993** Jing, 2016 Moolchandani, A. Kumar, and Sarangi, 2021

References

Adiono, Trio et al. (2018). "Design of Neural Network Architecture using Systolic Array Implemented in Verilog Code". In: *2018 International Symposium on Electronics and Smart Devices (ISESD)*, pp. 1–4. DOI: 10.1109/ISESD.2018.8605478.

Berchmans, Deepa and SS Kumar (2014). "Optical character recognition: an overview and an insight". In: *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. IEEE, pp. 1361–1365.

Jing, Yuan (2016). "An Efficient FPGA Implementation of Optical Character Recognition System for License Plate Recognition". In.

Leiner, Barba J. et al. (2008). "Hardware Architecture for FPGA Implemetation of Neural Network and its Application in Images Processing". In: *2008 4th Southern Conference on Programmable Logic*, pp. 209–212. DOI: 10.1109/SPL.2008.4547759.

Moolchandani, Diksha, Anshul Kumar, and Smruti R. Sarangi (2021). "Accelerating CNN Inference on ASICs: A Survey". In: *Journal of Systems Architecture* 113, p. 101887. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2020.101887. URL: https://www.sciencedirect.com/science/article/pii/S1383762120301612.

Xiao, Rui, Junsheng Shi, and Chao Zhang (2020). "FPGA implementation of CNN for handwritten digit recognition". In: *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. Vol. 1. IEEE, pp. 1128–1133.

Yogatama, Bobbi Winema et al. (2018). "FPGA-based Optical Character Recognition for Handwritten Mathematical Expressions". In: *2018 International SoC Design Conference (ISOCC)*. IEEE, pp. 42–43.