Employing Software and Hardware level Approximations for the optimization of Deep Neural Networks

Author

Rimsha Tariq

00000319467

Supervisor

Dr. Sajid Gul Khawaja

Co-Supervisor

Dr. Farhan Hussain

DEPARTMENT OF COMPUTER ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

APRIL 2022

Employing Software and Hardware level Approximations for the optimization of Deep Neural Networks

Author

Rimsha Tariq

0000000319467

A thesis submitted in partial fulfilment of the requirements for the degree of

MS Computer Engineering

Thesis Supervisor

Dr. Sajid Gul Khawaja

Thesis Supervisor’s Signature: ___________________________

DEPARTMENT OF COMPUTER ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, ISLAMABAD

APRIL, 2022

Declaration

I certify that this research work titled “Employing Software and Hardware level Approximations for the optimization of Deep Neural Networks” is my work. The work has not been presented elsewhere for assessment. The material that has been used from other sources has been properly acknowledged/referred.

Signature of Student

Rimsha Tariq

319467

Language Correctness Certificate

This thesis has been read by an English expert and is free of typing, syntax, semantic, grammatical, and spelling mistakes. The thesis is also according to the format given by the university.

Signature of Student

Rimsha Tariq

319467

Signature of Supervisor

Dr. Sajid Gul Khawaja

Copyright Statement

Acknowledgements

All praise and glory to Allah Almighty the kindest and most merciful, for being my strength and providing me with the knowledge and courage to carry out and complete this grueling thesis work.

I would like to express gratitude to my advisor Dr. Sajid Gul Khawaja and the entire Guidance committee Dr. Muhammad Usman Akram, Dr. Arsalan Shaukat, and Dr. Farhan Hussain for their guidance and for encouraging us to work hard and smart. I have found them very helpful while discussing the optimization issues in this dissertation work. Their critical comments on our work have certainly made me think of new ideas and techniques in the fields of optimization and software simulation.

Finally, I would like to express my gratitude to my parents, friends, and siblings who kept backing me up all the time both financially and morally. Without them, it would be impossible for me to accomplish this task.

Abstract

Neural Networks (NNs) are the core algorithms for many complex Artificial Intelligence (AI) applications, such as image and video classification and recognition, signal processing, etc. But these algorithms are both memory and computationally exhaustive, making it hard to deploy them on systems with restricted hardware sources. Subsequently, these systems are also extremely power greedy and expect a major amount of energy resources to perform the required computations. Approximate Computing (AC) has been gaining prominence for relieving computational and memory requirements of Deep Neural Networks (DNNs) benefiting from their error tolerance behavior. AC can be separated into two types of Hardware and Software layer approximations. In this research, we have considered optimization for CNN algorithms for H/W platforms specifically FPGAs. In this regard, we proposed multiple automated tools. The first tool deals with the memory optimization at S\W level approximations estimating the best levels (N) to quantize and encode weights with respect to user-defined requirements based on a genetic algorithm (GA). The GA makes use of a regression equation to determine the best population. This proposed module was tested for VGG-16, Cifar-Quick, and LeNet-5 returning the final population with an absolute maximum error of 0.038 when tested for originally quantized weights. Further, we proposed the design of an efficient decoder based on Canonical Huffman that can be utilized for the efficient decompression of weights in CNN. The proposed design makes use of Hash functions to effectively decode the weights eliminating the need for a searching dictionary. The proposed design decodes a single weight in a single clock cycle. Our proposed design has a maximum frequency of 408.97MHz utilizing 1% of system LUTs when tested for the Aritix 7 platform. Lastly, the third module deals with the estimation of the best approximate multipliers for H/W. The proposed module is based on GA and utilizes the tf-approximate library to calculate the accuracy loss in models for approximate multipliers. It was noted that the complex CNN model VGG-16 required more iterations to determine the best multipliers compared to the simpler model such as LeNet-5.

Table of Contents

List of Figures

List of Tables

Chapter-1 Introduction

The prodigious success of machine learning has led us into a period of artificial intelligence. Deep Neural Networks are the most pre-eminent algorithms in machine learning, used to solve many complex problems including natural language processing (NLP), computer vision, and image analysis. The presence of a wide dataset in cloud platforms and accessibility of tremendous computing sources (e.g., GPUs) are significant factors behind this achievement. Henceforth research on deep neural networks has gradually increased and many architectures have been proposed. The graph in Figure1 is taken from google trends showing the global interest in DNN in recent years.

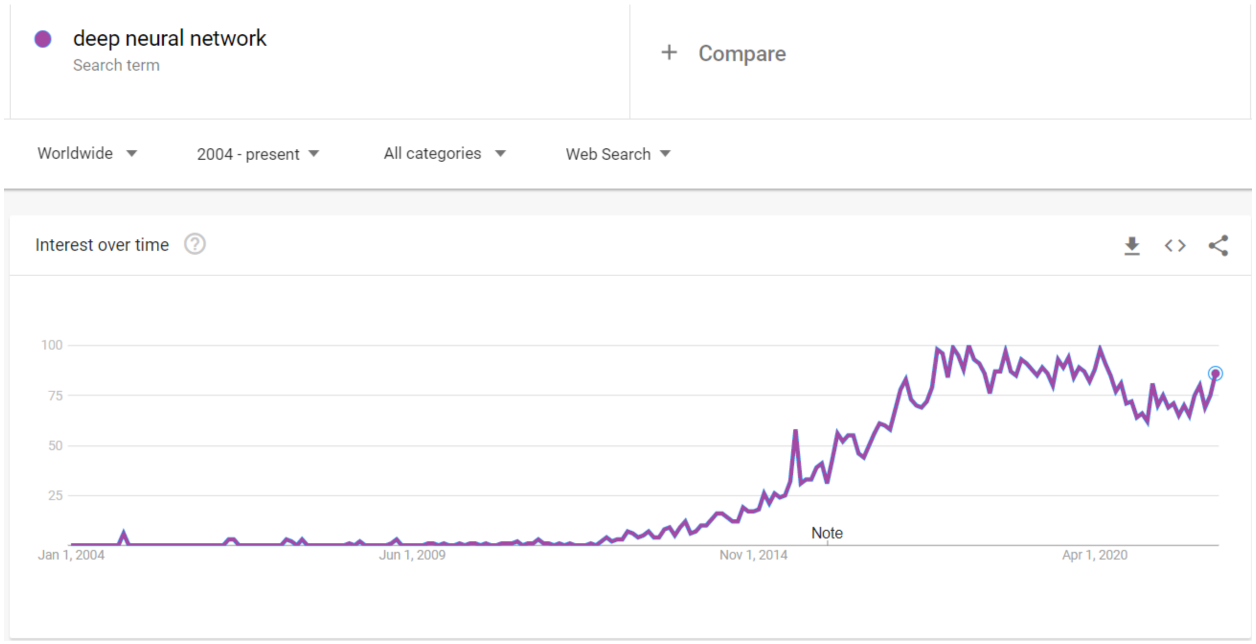


Figure 1: Interest in DNN with respect to time [1]

Accuracy is considered the most fundamental metric of many CNN models at the expense of more parameters, more layers, and more operations which result in high complexity of the model. The graphs in Figures 2 and 3 represent the depth and parameters of the most popular Deep neural networks being used in recent years for image classification and other applications of Artificial Intelligence.

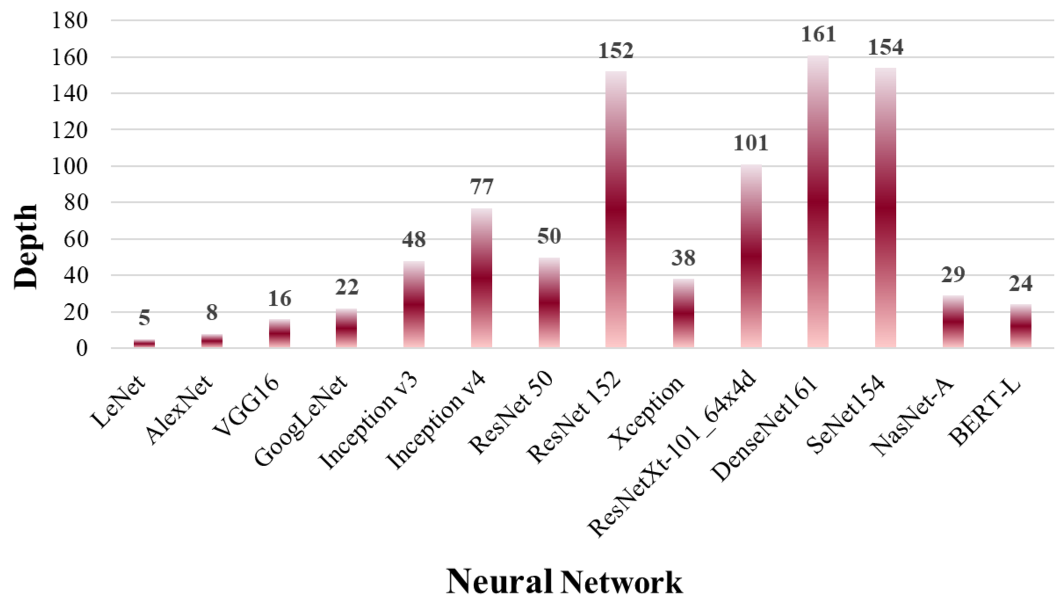


Figure 2: Depths (Layers) in DNN [2]

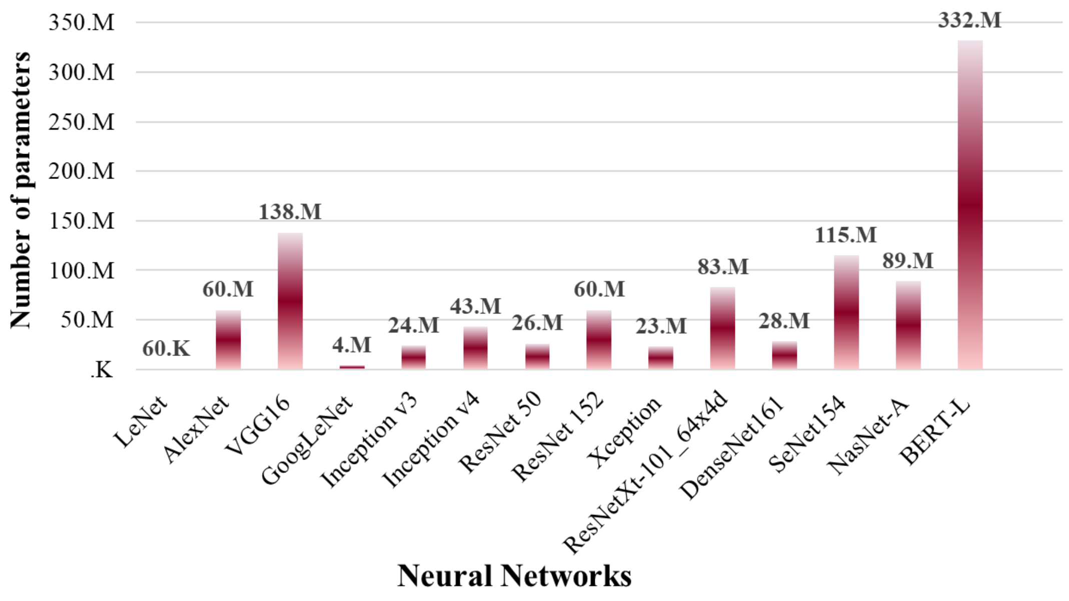


Figure 3: Number of Parameters in mostly used DNNs [2]

With the evolution of technology, embedded systems have become the main building blocks for technology using edge-computing such as IOTs. CNNs are more frequently trying to install on embedded systems and cellular devices. In recent years, FPGA (Field Programmable gate array) technology has illustrated strong capabilities in implementing embedded intelligence. At the same time, chip memory limits are also a bottleneck for FPGA, so managing available system resources to achieve high performance is also one of the challenges for researchers. Due to these restrictions, CNNs models are pre-trained offline and are implemented in these resource-scarce embedded systems for the inference stage.

Despite that, FPGA having limited on-chip memory is still problematic, it relies on external off-chip memory (DDR RAMs connected to FPGAs) to store weights of pre-trained CNN models. However, frequent access to off-chip memory means higher power consumption and latency resulting in performance reduction. For complex CNN's on-chip memory is infeasible for computations and too many buffers are required even after optimization leading to the use of external memory [3]. Convolution computation in the inference stage is still computationally exhaustive, and resource-intensive as it requires a large number of multiply-accumulate (MAC) operations. Moreover, the number of multiplication operations gets higher while tackling complex problems like object recognition and detection. Multiplication operations are always the most difficult to implement and are time-consuming in calculations. Thus, research on accelerating neural networks has steadily increased and several hardware and software methods have been proposed for the better performance of CNN models in FPGA.

Recent research shows that approximate computing (AC) has proved to be an effective technique for the acceleration and compression of neural networks. It involves such computational methods that return a potentially inaccurate result instead of an accurate result, reducing computations and memory requirements with minimal effect on the accuracy of the algorithm. These methods will be discussed in detail in Chapter 2.

Problem Statement

Approximate computing is the preferred approach for developing power-efficient systems with better execution times and saving computational sources. These techniques can be classified into two types software level (S/W) approximation and hardware level(H/W) approximation and can be applied on different levels of the neural network. In this research, the challenge aimed is to develop an automated tool for estimation of the best approximate components to optimize the performance of CNNs in Hardware(H/W) platforms specifically in FPGAs on basis of accuracy, power, energy, and space.

Aims and Objectives

The main objective of this thesis research are as follows:

Approximate Computing

1. Development of a framework to select the best approximate parameters for the quantization for efficient memory optimization in CNN models.
2. Design a framework to select the best approximate multipliers for CNN algorithms.

Hardware Decoder

To develop an efficient hardware decoder for weight compression having the following properties:

- Decoding of single weight within a single clock cycle
- Utilizing a lesser number of LUTs as compared to previous work
- To overcome searching in LUT at decompressing stage.

Structure of Thesis

This thesis work is structured as follows:

Chapter 2: Covers the brief literature review of signification work done by researchers in the past few years for the optimization of neural networks.

Chapter 3: Methodology and experimental results for Tool designed for optimizing CNN with S/W Approximation Techniques.

Chapter 4: Methodology and experimental results for Hardware decoder for data decompression in FPGAs.

Chapter 5: Methodology and experimental results for Tool designed for optimizing CNN with H/W Approximation Techniques.

Chapter 6: Concludes the thesis and defined the future scope of this research.

Chapter-2: Literature Review

The proposed work in this thesis is based on methods used for efficient utilization and implementation of CNN models in FPGAs such as Approximate Computing for Machine Learning and Reconfigurable architectures for memory space optimization in hardware.

In order to understand the basic idea concerning these topics, our literature is conducted in three parts. In the first part, we have reviewed different approximate computing techniques for CNNs. The second part reviews Reconfigurable hardware architectures specifically designed for decompressing data, compressed for memory optimization in FPGAs. While the last part reviews the tools designed to estimate the best approximate techniques in CNNs.

2.1 Approximate Computing techniques

AC is primarily used for error-tolerant applications such as deep learning, signal and image processing, etc. The approximations can be applied at various levels of the CNN and are divided into two types i.e., software-level approximation and hardware-level approximations.

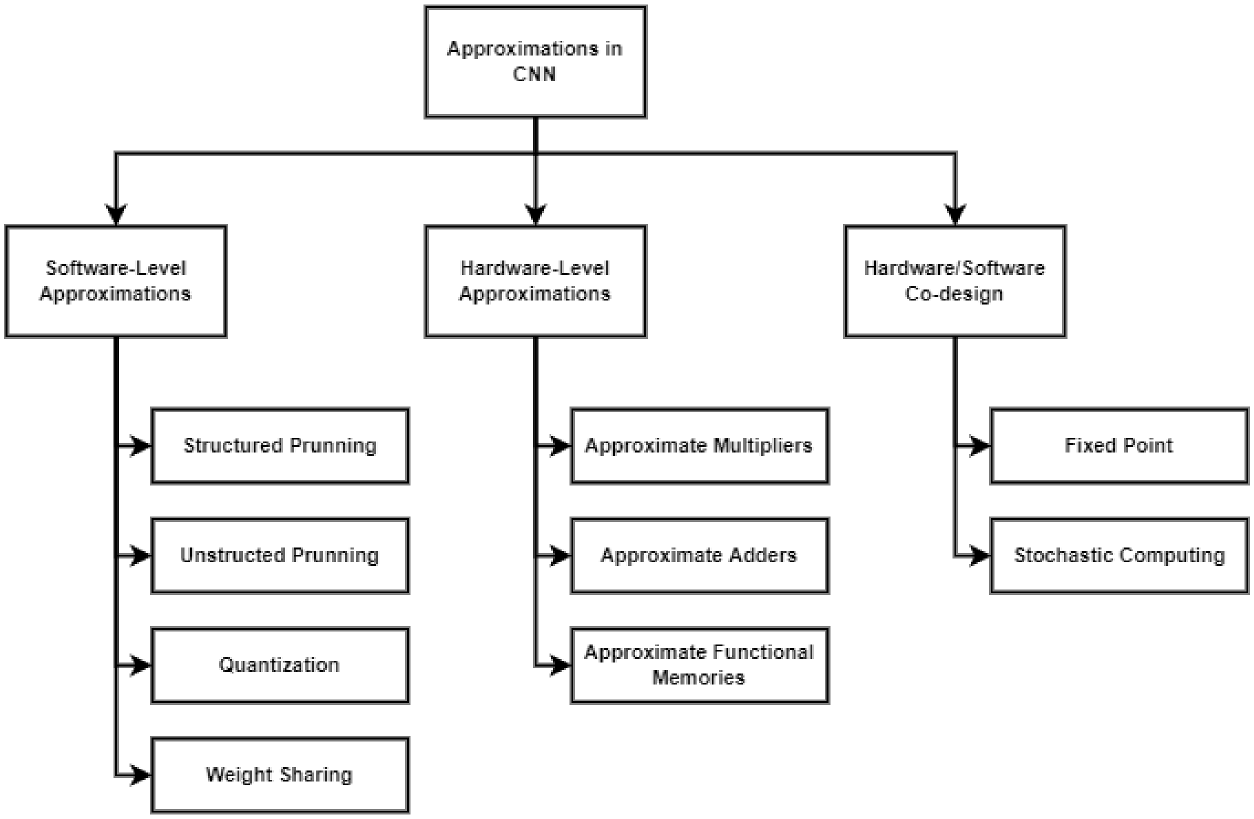


Figure 4: Approximate Computing Techniques for CNNs [4-15]

Research has also used software-hardware co-design approximation techniques for high precision data. Figure 4 shows the AC techniques employed to accelerate the performance of CNNs based on the layers of abstraction.

2.1.1 Software-Level Approximations

The software level approximations skip or lesson the computations and memory requirements in CNNs to improve overall performance keeping significant accuracy. Weight sharing, Pruning, and Precision Reduction are used for computation reduction. It allows network re-training which enables the network to regain its lost efficiency. For Further optimization memory at hardware, post-training quantization followed by weight-encoding is performed. This reduces the size of weights to store in FPGA memories. This technique is generally known as data compression. The software level approximations are mostly applied at the structural level the of CNN algorithm.

For this research project, we have used post-training quantization and weight-encoding for memory optimization of CNNs in FPGAs.

Hao et al [4] recommend a novel method of removing unnecessary filters in the convolutional layers with a small weight magnitude compared to a threshold. This threshold is determined using standard deviation of weights in a processed layer. This pruning technique proved to be useful giving negligible loss in accuracy. They achieved 34% and 24% of FLOP reduction by pruning some convolution layers of VGG and ResNet50/110 by 50% when tested on the CIFAR-10 dataset. Song Han et al[5] introduced a deep compression method with three stages, pruning, trained quantization, and Huffman encoding. Quantization in second stage is employed using the k-means algorithm. Quantizing weights in n-clusters. While pruning of network is done using [4]. The method was able to reduce the size of AlexNet from 240MBs to 6.9 MBs. VGG-16 from 552MB to 11.3MBs. But due to Huffman coded weights the model is unable to perform efficiently for general-purpose processors. To overcome these limitations the authors proposed a special hardware at [6] serving as an accelerator for these networks.

Alqahtani et al. [7] proposed a majority voting technique comparing the activation values among neurons and assigning them a quantitative score to calculate the importance of the neuron. The model is pruned during the training phase eliminating the need for fine-tunning and pre-training mechanisms. They pruned 79% of model parameters for CIFAR-10 with no loss in accuracy and 91% of the model was pruned in case of MNIST with no loss in efficiency of the system.

Jacob et. al. [8] suggested a quantization procedure, using integer arithmetic at the inference of the CNN model for computations. The integer arithmetic requires a lesser number of bits and provides higher efficiency compared to floating-point operation reducing the computation complexity in the model. To maintain the original accuracy, the authors designed a training procedure to resolve the tradeoff between latency and accuracy. They achieved 3% of accuracy loss with ReLU6 activation and 4% with ReLU on Inception V3 for ImageNet classification.

2.1.2 Hardware-Level Approximations

The hardware-level approximations help the network by easing computations in the model. Approximate Multipliers and Adders are used in CNNs for making approximations. These approximations result in a non-recoverable loss in accuracy. Due to the error-resilient nature of CNNs, these approximations are proved effective and reduce area, power, and computations with minimal loss in accuracy.

V.Mrazek et al [9] designed approximate adders and multipliers library called EvoApprox8b. This library comprises 430, 8-bit approximate adders which are generated from 13 conventional adders, and 471 8-bit signed approximate multipliers generated from 6 conventional multipliers. These approximate units were generated using Cartesian genetic programming. Authors have examined these approximate units using Cadence Encounter RTL Compiler and TSMC 180 nm and 45nm library and provided seven Petro error metrics and parameters such as area, power, and delay for both libraries.

Z. Wang et al [10] suggested an approximate high-speed execution of convolution layers in the CNN model, using Approximate Multiply-Accumulate Array (AMAs). Mitchell’s algorithm [11] is utilized to produce AMAs along with an error correction unit. These AMAs are individually parameterizable and are connected in a systolic framework. They tested their work Sobel filter used for edge detection. The proposed work gave the maximum frequency of 206.8MHz compared to the original architecture with a maximum frequency of 184.5 MHz. But still, AMAA gives 13.3% of error.

J.Faraone et al [12] designed a family of RCCMs to minimize data loss from quantization. For this, the coefficients in RCCMs are made constant, and input is multiplied by the fixed coefficient using multiplexers or bit shifts so that the multiplier is highly optimized for FPGA implementation. They tested their results using AlexNet, ResNet10, and ResNet18. Results show that this approach achieves better accuracy than networks that constrain weight parameters to have binary or ternary values; while allowing the expensive multipliers usually used in fixed-point implementations to be replaced by shifters, adders, and small MUXes. Furthermore, the restricted number of possible coefficient values allows an encoding scheme to significantly reduce weight storage.

Many other approximate units are being developed some of them are presented in Table 1.

Table 1 Approximate Hardware Units designed for CNNs specifically

Authors	Approximate Unit	Network Structure	Implementation	Accuracy	Results
Kin et al [11], 2015	Approximate Mitchell's log multiplier	AlexNet	32 nm digital standard cell library, synopsis design complier	84.87 %	Proposed design saves 80% of energy compared to 32-bit fixed-point multiplier
Kin et al, [13], 2018	Approximate Mitchell's log multiplier	LeNet CudaConvNet	32 nm digital standard cell library, synopsis design complier	99.02 % 81.43%	76.6% power reductions compared to fixed-point multiplier
Kowsalya et al [14] 2019	Pipelined hybrid merged adders	-	Xilinx Virtex 7 FPGA	-	Area and power consumption are reduced by 50%
Luo et al, [15] 2019	Single Clock Cycle Adder	LeNet	65 nm CMOS technology	98.7%	Speed up increased by 2.8x and 59.9% reduction in PDP
Chuliang et al, [16],2019	Reconfigurable approximate multiplier	VGG-16	Xilinx ZCU102 FPGA	-	17% and 15% reductions in latency and power

2.1.3 Hardware-Software Co-design Approximations

The Hardware-Software Co-design approximation helps decrease the computational loads and memory requirements by reducing the precision of the data within the network. Using of Fixed-point representation of data makes computations easier, and it decreases the complexity of computational units such as adders and multipliers. While Stochastic Computing reduces the hardware cost and enhances the power efficiency of the system. It is a low-cost approximation that utilizes logic gates such as XOR, OR, AND to accomplish multiplication while additions are achieved using multiplexers.

Ahmed et al. [17] designed an 8-bit fixed point parallel MAC unit using VHDL customized for FPGAs for the acceleration of CNNs specifically. The proposed unit achieved a high computation speed with a maximum frequency of 834.03 MHz implemented on Arria 10 and 594 MHz for Kintex-7

Tianchan et al [18] presented a novel method that utilizes binary-weight NN (BNN) training. This method provides data storage recycling, and incremental training, which leads to more efficient use of on-chip data storage for storing weights in FPGAs platform, this resulted in less access to off-chip, allowing the model to train with 14× lesser latency a neural network when compared to the conventional BNN training method.

2.2 Real-Time Hardware Decoder

In the past few years, several hardware accelerators have been proposed for the efficient implementation of DNNs. However, only a few of these methods apply data compression techniques. Typical solutions for FPGA concentrate on the optimization of computations for effective processing time. In contrast to the approaches proposed for the compression of DNN addressing limited recourses in embedded devices discussed below.

Huang et al. [19], proposed a novel algorithm for training pruning agents, deleting redundant convolutional filters with minimal effect on the accuracy. They demonstrated that pruning 78.4% of ResNet-18 convolution filters results in only a 2.9% of loss in accuracy. Aghasi et al. [20] proposed a post-processing method that involves layer-by-layer pruning of a trained neural network to maintain the internal responses.

However, pruning techniques provide lossy compression as they change the uncompressed structure of CNN affecting the network performance.

Dahri et al. [21] suggest the Huffman algorithm on FPGA using Xilinx ISE 8.2i. This technique cannot guarantee real-time decoding as the search for conducting codeword matching is done sequentially causing system starvation.

Mansour et al [22] suggest a Huffman algorithm applying a two-level Look-Up Table (LUT). Nevertheless, this method demands a large LUT and utilizes various repetitions of shortcode words which leads to increasing decoder memory space.

Lin et al [23] proposed a two-stage decompression procedure based on approximated adaptive Huffman and PDLZW. However, implementing this algorithm in real-time fails to ensure a weight decompression in a single clock cycle.

Malach et al [24] proposed novel compression and decompression techniques and a real-time Huffman decoder utilizing three LUTs and address generation unit generating address by adding offset and index of code in LUT for decompression in a single clock cycle. They achieved a 68.72% of compression ratio in AlexNet compared to7zip compression which achieves a 69.57% of compression ratio. However, they did not discuss implementation resources utilized for hardware decoder.

Aspar et al [25] proposed Parallel Huffman decoder module with an optimized LUT They designed Bit-Serial and Bit parallel decoders for Altera FPGA with maximum operating speed of 10.89 MHz and 11.54 MHz.

Angulo J et at [26] designed Huffman based decoder for the decompression of high-speed seismic data. The data was processed using DCT and uniform quantization and passed on to the decoder. They achieved maximum operating speed of 256MHz using less than 1% of FPGA resources.

Najmabadi et at [27] architecture for asymmetric numerical systems (tANS) entropy decoder and compared its efficiency with canonical Huffman decoder designed in Xilinx. They achieved the throughput of 125MHz for tANs while 50MHz for the canonical decoder The drawback this procedure is that each encoded symbol may take more than single clock cycle for decoding.

2.3 Real Tools for Optimizing Approximate computing techniques

Nogami et al [28] explored the numerical quantization and presented a unique “Variable Bin-size Quantization (VBQ)” representation. In this bin limits in quantization are optimized to achieve high accuracy in CNN model. For optimization purposes, authors make use of a genetic algorithm. The quantization process is done for inference purposes only. This reduced the computational cost to great extent. They determined a heuristic function for bin distribution and optimized the parameters of this function using simulated annealing. They used this function to find the parameters in AlexNet and VGG16. Results show that AlexNet and VGG16 had only 1.5% and 2% accuracy loss with 4-bit quantization in comparison with 32-bit floating-point. However, this method can be time exhausting as the updated variable bin size is used to quantize the original model and evaluate its performance with the fitness function.

Ansari et al [29] identified that introducing noise to AMs can improve the accuracy of NN. Then they intended to develop a predictor to estimate how well an AM is expected to work for a given CNN model. For this purpose, they extracted features from AMs that are likely to represent their working superiority with other multiplies. These feature values were based on 9 different error metrics extracted from 100 deliberately designed multipliers and 500 CGP-based multipliers. After selecting the best features, they used them to train their predictor a 3-layered MLP model. To verify the proposed predictor 114 AM were tested for their performance in LeNet-5 and AlexNet for SVHN and ImageNet datasets, respectively. The Final analysis made by the authors is classification accuracy depends on the nature of the application and depends on network type along with dataset, activation functions, and even compiler parameters. Hence features used to train the predictor cannot assure that the identified multiplier will give the best results for a certain application.

2.4 Research Gaps

From the previous work done in the field of approximate computing, it is noted that many software and hardware techniques proposed efficiently optimize CNN models with minimal loss in accuracy. However, to achieve best approximation with minimal accuracy loss these techniques are tested again and again with different input parameters i.e., time exhausting. To overcome this problem a tool can be designed for the automatic estimation of approximate components. In section

2.3 authors did design these tools but failed to ensue and specify the quality of these tools. Hence research gaps in case of approximate computing can be defined as:

- Lack of time-efficient tools for automatic estimation of approximate components in CNNs.
- Lack of techniques ensuring and specifying the quality of automatic tools for estimation
- Lack of techniques for efficient memory utilization in CNNs for hardware optimizations.

Furthermore, it was also learned that weight compression is the most pre-eminent technique for memory space optimization in FPGAs. However, these weights are to be decompressed while performing calculations leading us to study decompression in Hardware. Many hardware decoders have been designed in this regard mentioned in section 2.2, but these decoders are time-consuming and resource exhaustive. Following research gaps were observed in case of Hardware decoders

- Hardware Decoder requiring multiple LUTs for decompression.
- Hardware Decoder requiring multiple clock cycles for decoding a single compressed weight.

2.5 Chapter Summary

In this chapter we studied the previous work done in approximations of deep neural networks for optimization purposes. Many software and hardware techniques were noted and discussed. It was noted that one of the most important techniques used for memory optimization of CNN in FPGAs is data compression which includes pruning of the network followed by post training quantization of network weights and finally encoding of these quantized weights. This also led us to study the work done for the decoding of these quantized weights in FPGA. Many decoders have been designed but most of them utilizes more than a single clock cycle for decoding of single weight and some of them make use of multiple LUTs increasing the resource utilization. On other hand hardware approximations are done for the reduction of easing complex calculations by using approximate multipliers and adders. There are also hardware-software co-design techniques that make use of both methods for making approximations. Finally, we investigated the tools that can be used to predict the best parameters utilized to approximate a CNN model but the designed techniques lack in ensuring and specifying the quality of designed tools for estimation.

Chapter-3: Optimizing CNN with S/W Approximation Techniques for memory optimization

This chapter provides a detailed overview of our proposed tools for optimization of CNN in FPGAs. We have proposed multiple tools in this regard. The first tool deals with memory optimization at software level approximations where quantization and canonical Huffman coding have been used to compress pre-trained weights. The proposed tool estimates the approximate parameters at which a CNN model can be quantized for user defined “acceptable error” in accuracy. In order to implement a memory optimization scheme in Hardware, a canonical Huffman encoder is used to encode weights reducing memory size to store weights in an FPGA. This leads us to design our second tool, a novel architecture for canonical Huffman decoder in hardware that works within a single clock cycle with restricted FPGA resources. Finally, our third tool deals with approximate units, specifically multipliers. This tool identifies the set of best approximate multipliers based on "acceptable error”, “area” and “power” in FPGAs. The overview of proposed methodology can be seen in Figure 5.

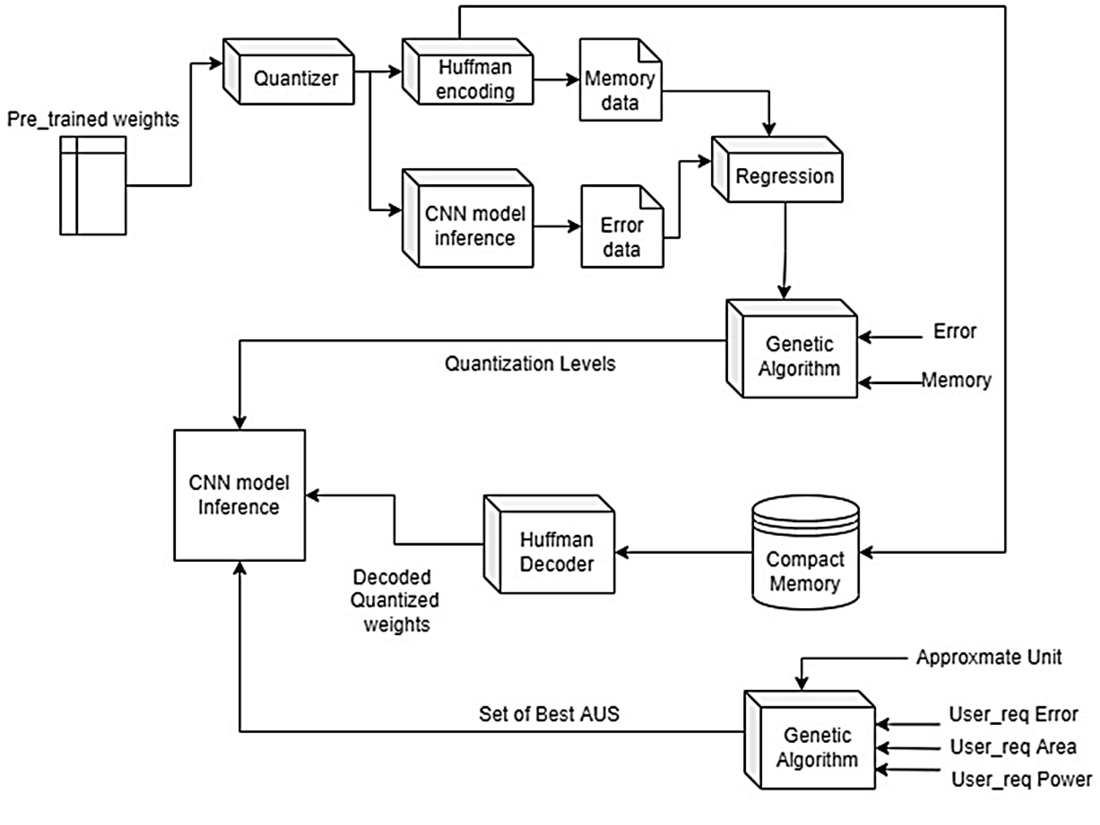


Figure 5: Overview of Proposed methodology

Both proposed automated estimation tools are based on genetic algorithms. John Holland created the genetic algorithm (GA), along with his colleagues between the 1960s and 1970s [30]. This algorithm is extracted from Charles Darwin's theory of natural selection in biological evolution. Since then, many variants of Genetic Algorithm have been developed for optimization problems in image processing, signal processing and artificial intelligence. This chapter discusses the Automated tool for the selection of best parameters in data compression for CNNs.

3.1 Automated tool for Data Compression in CNNs

The proposed Framework is specifically designed for data compression in CNNs. For this purpose, we collected data for different types of quantization methods and found a relation between accuracy loss that occurred after using quantized weights to N levels at which quantization is done, parameters and layers of CNN model. Similarly, we found a relationship of the said factors with memory of encoded weights. This relationship is determined using regression which is later used by a proposed automated tool. Hence, before jumping into the working of the main automated tool let us discuss the process of data collection and regression selection. The overall block diagram can be seen in the following Figure 6.

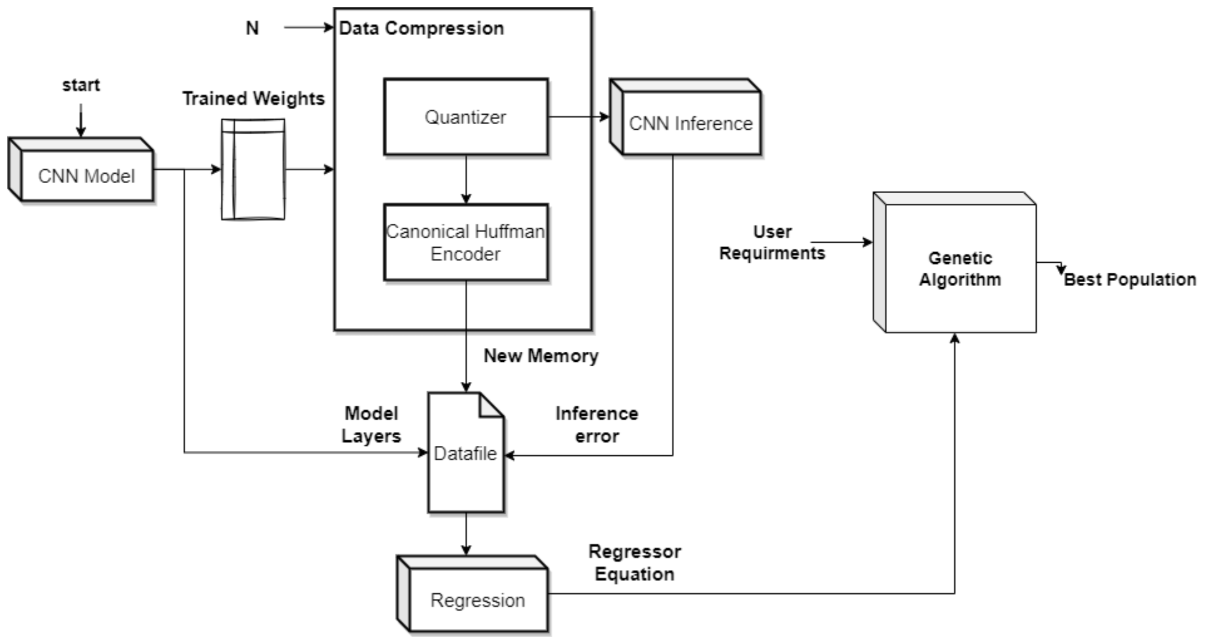


Figure 6: Overall Workflow of S/W Automated Tool

3.1.1 Data Compression

In our proposed framework Quantizer and Canonical Huffman Encoder has been used for the compression of data. The data compression model inputs “pre-trained weights” for a certain CNN model, N levels, and quantization type (Q-type).

3.1.1.1 *Quantization*

Quantization is a process of mapping continuous value to a set of discrete values. Quantizer is a post-training process that utilizes pre-trained CNN weights, N-levels and quantization type (Q-type) for the generation of codebook. This codebook is then used to quantize the weights.

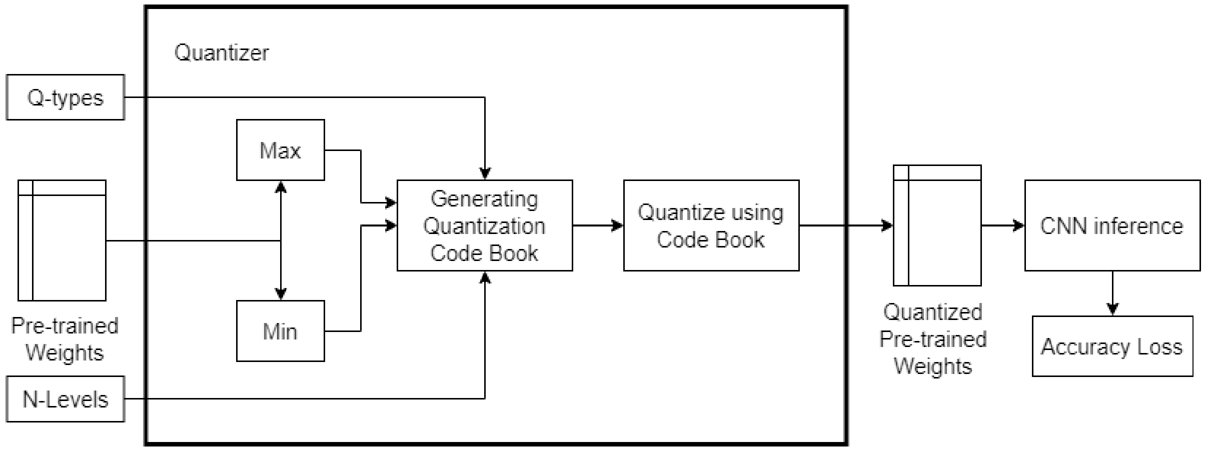


Figure 7: Workflow of Quantizer

Form Figure 7 it can be seen that quantized weights are been fed to CNN model for inference only and accuracy loss is being stored for different values of N. For our framework we have used three different types of quantization methods:

1. ***Uniform Quantization***

   In Uniform Quantization “N-levels” are evenly spaced between minimum and maximum value of weights. Code Book generation for uniform is done using equation 1 and 2.

$$Step_{size} = \frac{(Max - Min)}{(N-1)} \tag{1}$$

$$CodeBook(x_i) = \begin{cases} x = \min \quad , & i = 0 \\ x = x_{i-1} + step_{Size}, & 0 > i < N-1 \\ x = \max \quad , & i = N-1 \end{cases} \tag{2}$$

2. Non-Uniform Quantization

In Non-Uniform Quantization "N-levels" are unevenly spaced between minimum and maximum value of weights. Code Book generation for non-uniform is done through following formulas in equation 3 and 4.

$$Step_{size}(x_i) = \begin{cases} x = \dfrac{(Max - Min)}{(N-1)}, & i = 0 \\ x = Step_{size}(x_{i-1}) + \dfrac{Step_{size}(x_{i-1})}{i + m}, & i > 0 \end{cases} \tag{3}$$

$$CodeBook(x_i) = \begin{cases} x = \min, & i = 0 \\ x = x_{i-1} + step_{Size_i}, & 0 > i < N-1 \\ x = \max, & i = N-1 \end{cases} \tag{4}$$

3. Asymmetric Quantization

Asymmetric Quantization is done by using a zero-point (also called quantization bias or offset) in addition to the scale factor. This scale factor is determined using following equation 5.

$$Scale = \frac{\max(w) - \min(w)}{\max(Wq) - \min(Wq)} \tag{5}$$

Where W represents original weights value and Wq represents weights are uniformly quantized.

As for our work we used i^{th}quantization minimum and maximum value and i^{th} maximum and minimum values of original weights to determine the scale factor. To do so we determined the step size in case of uniform quantization and adding that step size with respect to "N" levels. We determined the 2nd minimum value and 2nd highest value in quantization using following formulas mentioned in equation 6 and 7.

$$Min_i = \min(W_{i-1}) + step_{size} \tag{6}$$

$$Max_i = \max(W_{i-1}) + step_{size} \tag{7}$$

While zero-point is calculated using equation 8 and codebook is generated using equation 9.

$$zero\ point(z) = min(Wq_i) - \frac{min\ (W_i)}{scale} \tag{8}$$

$$CodeBook(x_i) = \begin{cases} x = \ min(W) & , \quad i = 0 \\ x = z + \dfrac{W_i}{scale}, & 0 > i < N-1 \\ x = max(W) & , \quad i = N-1 \end{cases} \tag{9}$$

After codebook being generated the original weights are mapped to nearest values present in the codebook. Separate codebooks are generated for all convolutional and dense layers with respect to their weight parameters.

3.1.1.2 Variable Weight Encoding

Encoding is done to further reduce the size of memory after quantization in order to stored weights in FPGAs. From Figure 6 it can be seen that after encoding weights, we have calculated the new memory of our encoded weight file. Compression ratio and percentage of memory being saved for all values of "N-levels" were calculated using following equations 10 and 11 and stored in datafile.

$$Compression\ Ratio = \frac{Original\ Memory}{Compressed\ Memory} \tag{10}$$

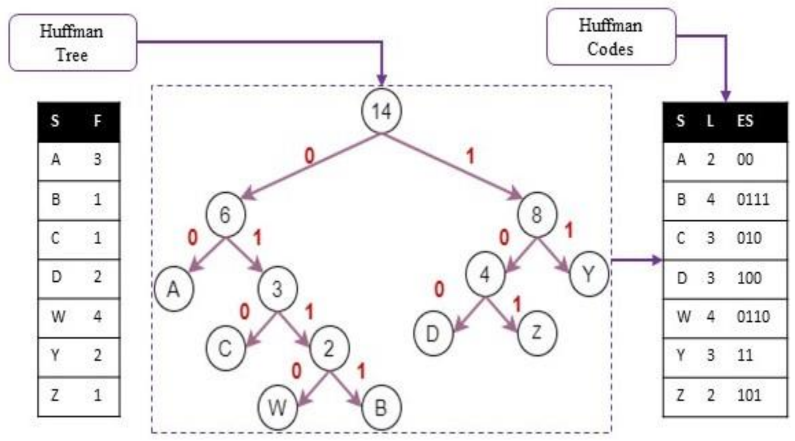
$$Percent\ Saved\ Memory = \frac{(|new_{memory} - original_{memory}|)}{original_{memory}} \tag{11}$$

It is noted that Quantizing weights with specified "N-levels" helped reduce the memory size to great extent after encoding. For example, if weights are quantized with N = 10 levels it means that the weight file will contain only 10 repeated unique symbols which can be encoded using codes having a lesser number of bits. Hence N is directly proportional to the average bits per symbol.

3.1.1.2.1 Canonical Huffman Encoding

In this research we have preferred canonized compression of weights as it proved to be beneficial for our real-time hardware decoder which will be discussed later in chapter 4.

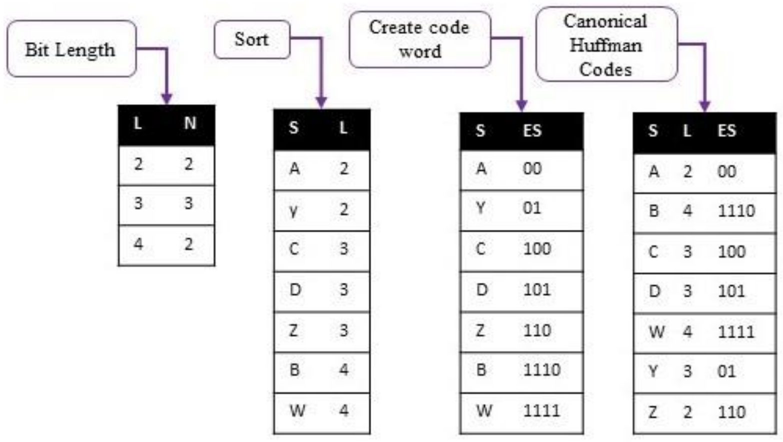
Canonical Huffman Coding is a subset of the Huffman Coding having numerical sequence property. To Canonize the weights, we must encode them with Huffman encoder to get the bit length information for each code assigned to all unique symbols (weights). Then to Canonize the Huffman codes first step is to sort the codes according to their bit lengths and then sort them lexicographically for the same bit lengths. Thus, the bit length for each code remains the same as Huffman code. Second step is to assign them new codes, the first symbol is assigned zero with the same bit length as its original code. For successive symbols get codes incremented by one if they have the same bit lengths as the previous symbol. If bit length changes, then after incrementing the previous code, zeros are appended to the right to match the bit length. An example of Huffman and Canonical Huffman code is shown in Figure 8.



S	F
A	3
B	1
C	1
D	2
W	4
Y	2
Z	1

S	L	ES
A	2	00
B	4	0111
C	3	010
D	3	100
W	4	0110
Y	3	11
Z	2	101

Figure 8(a): Huffman Tree Codes



L	N
2	2
3	3
4	2

S	L
A	2
y	2
C	3
D	3
Z	3
B	4
W	4

S	ES
A	00
Y	01
C	100
D	101
Z	110
B	1110
W	1111

S	L	ES
A	2	00
B	4	1110
C	3	100
D	3	101
W	4	1111
Y	3	01
Z	2	110

Figure 8(b): Canonized Huffman Codes

3.1.2 Approximate Data Collection

Data collection is done to test approximate behavior of the CNN model at inference level. This behavior is tested based on following two cases:

- Same Quantization levels (N) for both Convolutional and Dense layers
- Different Quantization levels (N) for Convolutional and Dense layers

Data for all quantization types (Q-types) discussed in previous sections is gathered for these two cases and attributes mentioned in Table2 were noted.

Table 2: Attributes Stored in Data file

Case 1: Different Levels for both Dense and Convolutional layers	**Case 2: Same Levels for both Dense and Convolutional layers**
Conv Levels (cN)	Levels (N)
Dense Levels (dN)	
Accuracy loss	Accuracy loss
Compression Ration	Compression Ration
Percent saved memory	Percent saved memory
No. of Parameters in CNN	Number of Parameters
No. of Convolution Layers in CNN	No. of Convolution Layers in CNN

To collect these attributes, following CNN models and Datasets were used shown in Table 3. These models and datasets are discussed in detail in section 3.2.1. These models were trained then quantized with different N-levels for all Q-types.

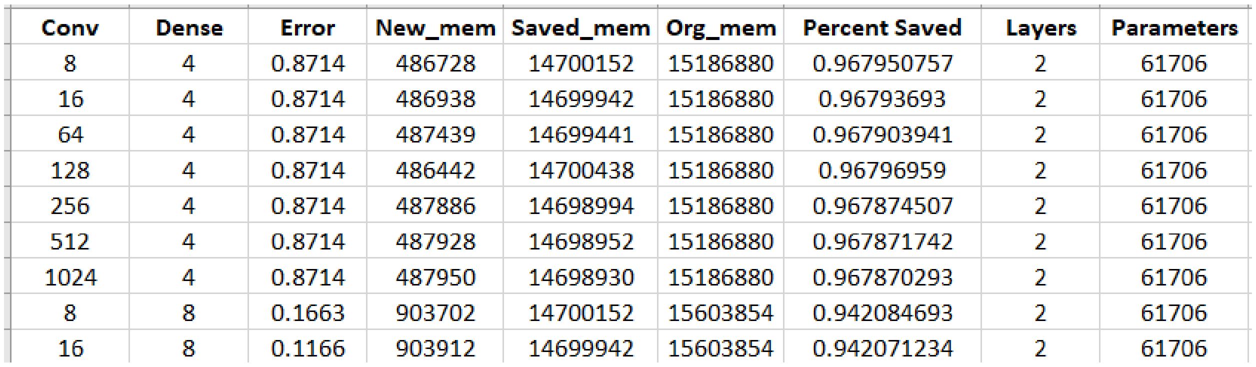
Table 3: CNN model used to Collect Data

Model	**Dataset**	**Layers**	**Parameters**	**Original Accuracy**	**Original Memory**
LeNet 5	MNIST	2	61,706	0.9849	120.36 KB
Cifar Quick	Cifar10	3	211,818	0.7302	847.272 KB
VGG16	ILSVR 2012	13	14,714,688	0.9389	129.33 MB

For different levels and Q-types different datafiles were created. Hence total 6 datafiles were created.

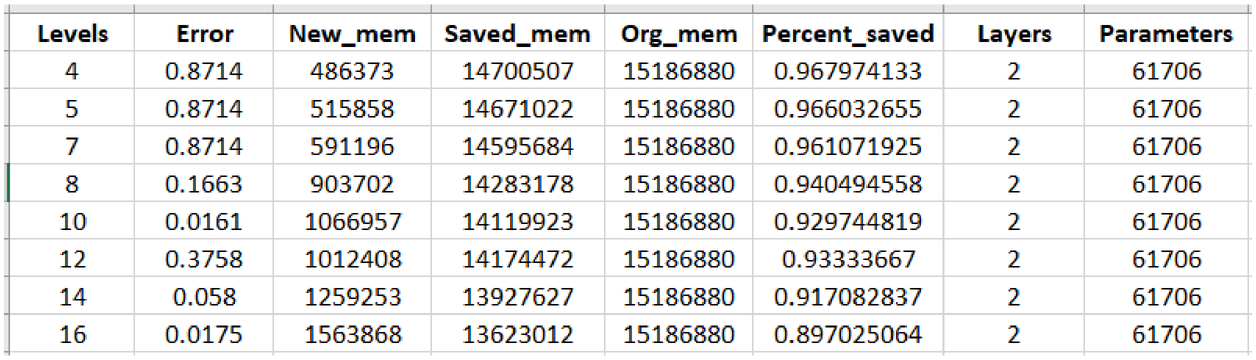
- Uniform Quantization [case1, case2]
- Non-Uniform Quantization [case1, case2]
- Asymmetric Quantization [case1, case2]

Visualization of Datafile is shown in Figure 9 and Figure 10



Conv	Dense	Error	New_mem	Saved_mem	Org_mem	Percent Saved	Layers	Parameters
8	4	0.8714	486728	14700152	15186880	0.967950757	2	61706
16	4	0.8714	486938	14699942	15186880	0.96793693	2	61706
64	4	0.8714	487439	14699441	15186880	0.967903941	2	61706
128	4	0.8714	486442	14700438	15186880	0.96796959	2	61706
256	4	0.8714	487886	14698994	15186880	0.967874507	2	61706
512	4	0.8714	487928	14698952	15186880	0.967871742	2	61706
1024	4	0.8714	487950	14698930	15186880	0.967870293	2	61706
8	8	0.1663	903702	14700152	15603854	0.942084693	2	61706
16	8	0.1166	903912	14699942	15603854	0.942071234	2	61706

Figure 9: Case 1 Different Levels for both Dense and Convolutional layers



Levels	Error	New_mem	Saved_mem	Org_mem	Percent_saved	Layers	Parameters
4	0.8714	486373	14700507	15186880	0.967974133	2	61706
5	0.8714	515858	14671022	15186880	0.966032655	2	61706
7	0.8714	591196	14595684	15186880	0.961071925	2	61706
8	0.1663	903702	14283178	15186880	0.940494558	2	61706
10	0.0161	1066957	14119923	15186880	0.929744819	2	61706
12	0.3758	1012408	14174472	15186880	0.93333667	2	61706
14	0.058	1259253	13927627	15186880	0.917082837	2	61706
16	0.0175	1563868	13623012	15186880	0.897025064	2	61706

Figure 10: Case 2 Same Levels for both Dense and Convolutional layers

3.1.3 Regression

Regression is used to find the relationship between continuous data. So, for our problem regression fits perfectly. So we tested the following regression on our data for memory and accuracy loss separately.

1. Linear Regression
2. Lasso Regression
3. Bayesian Ridge Regression

4. TheilSen Regressor
5. SVM Linear
6. SVM Polynomial
7. SVM RBF
8. SVM Sigmoid
9. Huber Regression
10. RANSAC Regression

To select the best regression equation, we developed an algorithm that calculates Mean Squared Error (MSE) for above-mentioned regressions tested for all datafiles. In order to do so, it inputs a Dataset and array of regressions to be tested. The dataset is divided in test and train parts. The following regression fit the training set and is tested on test dataset to obtain MSE using equations 12 and 13 mentioned in Algorithm 1

Algorithm 1: Selection best Regression Equation

```
Input: Datafile, FR (Regressor function array)
Output: out[Ft, MSE]
Procedure:
        t ← 0
        Re ← Rm ← $\varphi$  # Re: regressor error
                             # Rm: regressor memory
        train_m, test_m ← split(shuffle(Datafile))
        train_e, test_e ← split(shuffle(Datafile))
        While (t == length(FR) -1) do:
                Eq1 ← FRt(train_e) # for error
                Eq2 ← FRt(train_m) # for memory
                Ye = Eq1.predict(test_e)
                Ym = Eq1.predict(test_m)
                E ← $\frac{1}{n}\sum_{0}^{n}(Ye\ ,\ Test\_e)^2$          (12)
                $M \leftarrow \frac{1}{n}\sum_{0}^{n}(Ym\ ,\ Test\_m)^2$        (13)
                Re U E U FRt , Rm U M U FRt
        End
        FR error ← Min(Re , E)
        FR mem ← Min(Rm , M)
        Return FR error , FR me
```

To select the best regression for accuracy loss and memory percentage, MSE obtained for all datafiles were averaged and the regression with minimum average MSE is selected. Graphs for Average MSE for both accuracy loss and Memory loss are shown in Figure 11 and 12.

From Figure 11 and 12 it is noted that the lowest MSE for accuracy loss is given by SVM with kernel RBF while for memory percentage it is given by linear regression.

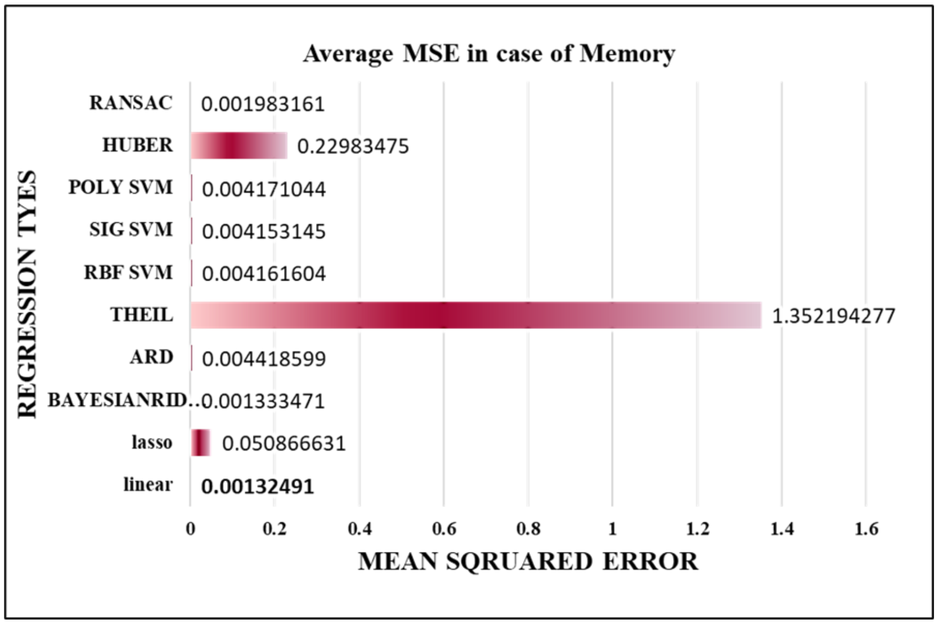


Figure 11: Average MSE for regressions for accuracy loss

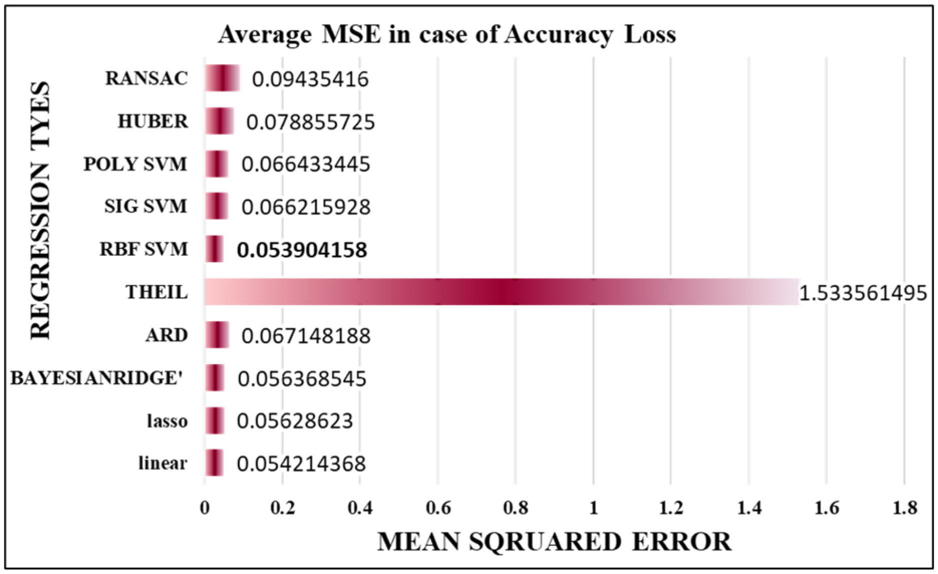


Figure 12: Average MSE for regressions for percentage saved memory

Now that we have calculated all input parameters required by our main module based on GA. Let us discuss the Framework of our main automated tool used for the estimation of parameters needed by quantization with respect to its user requirements.

3.1.4 Proposed Automated Tool

As discussed above the proposed automated tool is based on GA which makes use of inputs provided by users, to estimate the best parameters for the optimization of CNN model.

3.1.4.1 Inputs needed by Genetic Algorithm

Inputs required by proposed optimization tool are listed below:

1. Population Size required to initialize population (**size**)
2. User acceptable error in accuracy of CNN model (**er**)
3. Percentage Memory to be saved for CNN model (**mr**)
4. Number of generations (**gen**)
5. Threshold (**thresh**)
6. Preference variables (**α, β**)
7. Regression Equations for accuracy loss and memory percentage (**reg_e, reg_m**)
8. CNN parameters and layers (**par, lay**)

3.1.4.2 Output provided by Genetic Algorithm

The proposed algorithm provides following outputs:

1. Fitness of final population (**MSE**)
2. Fitness of each individual in final Population (**Find**)
3. Final Population (**Po**)

3.1.4.3 Working of Proposed Framework

The workflow of our proposed algorithm is presented in Algorithm 2. At the first iteration, it will use the population initialization function to initialize the population. Once the population is initialized it is passed to the fitness function. Then the said population is modified with respect to their individual fitness values in the population modification stage. In order to terminate the GA two of one condition must satisfy i.e., either the MSE value of the current population is lesser than the user-defined threshold or the number generation also known as loop iterations are completed.

If these conditions are not satisfied the GA will keep on working. The three sub-modules used in Algorithm 2 are explained in the next sections..

Algorithm 2: Pseudocode of Proposed Automated Tool (Genetic Algorithm)

```
Input      : size, er, mr , gen, thresh , α, β , reg_e ,  reg_r , par, lay
Output     : F_ind, Mse, Po
Procedure:
        t ← 0
        Po ← initialize(size)
        While(t < gen) do:
        F_indt,Mset ← Fitness(Po , er , mr , α, β , reg_e , reg_r , par, lay, )
        if(Mset <= thresh)do:
          break
        Pot ←  Modification(F_ind, Pot)
        End
        Return  F_ind, Mse, Po
```

3.1.4.4 Population Initialization

The population initialization procedure is called only once at the beginning of the genetic algorithm to initialize a random population of individuals to enable processing in GA. The size of the population (m) is inputted by users according to their needs. The size of the population defines the number of individuals present in the population. The population is initialized with respect to the type of problem the user is dealing with. In our situation, we want to determine the values of the best parameters for quantization i.e., "N-levels" representing an integer value. It has been discussed in Table 3 that N-levels are defined differently using two cases. Hence the population of random integers with size population size (m) has been generated for these two cases differently as represented in Table 4.

Table 4: The Population inilization array format for two defined cases

Case 1: Different Levels for both Dense and Convolutional layers	Case 2: Same Levels for both Dense and Convolutional layers
Po = array ([c1, d1], [c2, d2] ……, [cM, dM])	Po = array ([cd1], [cd2] ……., [cdM])

Where c represents the levels in convolutional layers and d represents the levels in dense layers

3.1.4.5 Fitness Evaluation

A fitness function is a specific type of objective function that determines how close a particular population is to user requirements using a single fitness value. For this purpose, it makes use of the selected regression equation in section 3.1.3. The accuracy loss and memory reduction percentage are determined using these regression equations for the inputted population. Fitness values of each individual member in the population and Fitness for the entire population are calculated. Based on these calculated values, the population is modified, i.e., a new population is generated. Fitness is calculated for all generations of the population until GA is terminated. Algorithm 3 is designed to calculate the fitness in the proposed tool.

Algorithm 3: Fitness Evaluation

```
Input: Po, er, mr, Svmrbf , lm, α and β
Output: FP, F_ind
Procedure:
    t ← 0
    F_ind ← $\varphi$
    Ye = lm.predict(Po)
    Ym = SVMrbf.predict(Po)
    E ← $\frac{1}{n}\sum(Ye\ ,\ er)^2 \times \alpha$ #MSE                          (14a)
    M ← $\frac{1}{n}\sum_0^n(Ym\ ,\ mr)^2 \times \beta$ #MSE                        (15a)
    FP = E+M
    While(t < length (Po)) do:
        # Absolute error of each individual
        Er ← abs( Yet – Ep) × α                                   (14 b)
        Mr ← abs(Ymt – Mp) × β                                   (15 b)
        F_ind U (Er + Mr)
        t ← t + 1
    End
    Return FR error, FR me
```

The proposed Fitness function in Algorithm 3 outputs a single fitness value for the entire population and individual fitness values for all members within the population. The input of fitness function is population, regression equations, user acceptable accuracy loss (er), user required saved memory (mr), and preference variables (α and β)

- **Preference Variables**

Preference variables are used to tell the proposed algorithm which user-defined parameter is critical. In the proposed algorithm **α** is assigned to accuracy loss, while **β** is assigned to the percentage of memory to be saved. To understand the use of the preference variables let's consider an example**,** if accuracy loss is more critical than the percentage of memory to be saved then the weight given to **α** will be greater than **β.** The sum of **α** and **β** is **1**.

Therefore, if both user-required parameters are equally critical then both **α** and **β** are assigned a value of 0.5.

Now that we know the inputs and outputs of the fitness function let's understand its internal working form Algorithm 3. Firstly, accuracy loss and percent saved memory has been determined using SVM with RBF kernel and linear regression equations calculated in section 3.1.3. Then using equation 14a and 15a provided in Algorithm 3, MSEs is determined for accuracy loss (E) and percent saved memory (M) with user-entered parameters (er and mr). The final fitness value (FP) of populations is the sum of both MSEs (E and M).

For estimating individual fitness for all members within a population, Absolute error (AE) is used defined in equations 14b and 15b in Algorithm 3 The predicted accuracy loss and saved memory for each individual is processed through the said equations and the answer is saved in (F_ind) array with respect to their indexes in population. This individual Fitness (F_ind) is used in the population modification process for updating the current population.

3.1.4.6 Population Modification

Population Modification is a process of selecting two or more individuals from the current population to generate new offspring and updating them into current population using following three procedures:

1. **Parent Selection**

   The proposed algorithm uses Fitness based selection sub-type of the survivor-based selection procedure. In this process, the individuals that have the least fitness values i.e., individuals with the largest (AE) are selected and passed over to the next process. Firstly, individual population fitness (F_ind) is copied to another array (F_n) to avoid any data losses. Secondly, the individual with the highest (AE) is determined and its index position is saved to the index array (indx) and the individual at that index in the population is saved to the selected parent list (Ps). In the next step, the individual fitness of the selected individual in the array (F_n) is set to the maximum integer value. The process is repeated to find the next parent and can be seen in Algorithm 4..

Algorithm 4: Parent Selection

```
Input: Po, F_ind
Output: Parents (Ps) , indexes (indx)
Procedure:
        t ← 0
        No. of parents = 2
        Ps← $\varphi$
        Indx ← $\varphi$
        F_n = copy(F_ind)
        While(t < No. of parents)) do:
                index ← max (F_n)
                indx U index
                Pst = Po[index]
                F_n[index] = -999999999
        End
        Return Ps , indx

        # Ps = Array (Parent1 , Parent2)
```

2. Crossover

In Crossover selected parents are used to generate one or more off-springs using the genetic material of the parents. For this process selected parents (Ps) are binarized with the same bit length. In this thesis, the one-point crossover is performed for generating new offspring, A random index within the bit length of binarized parents is produced. For generating the first offspring the first part of parent 1 in Ps is joined to the second part of parent 2 after that random index. The same procedure is repeated for generating the second offspring but with parent 2 as the first part and parent 1 as the second part. These newly generated offspring should be different from all individuals present in the current population if any of the offspring overlaps with the current population. Then that offspring is passed to the mutation process to generate new offspring. Once these offspring are finalized, they are converted to the decimal point and replaced with selected parents (Ps) using their index values (indx). This process is represented in Algorithm 5.

Algorithm 5: Crossover

```
 Input: Pst
Output: Offsprings
Procedure:
        t ← 0
        Offspring← $\varphi$
        #P1 ← Pst [0]
        #P2 ← Pst [1]
        While(t < No. of Parents)) do:
              cross_point ←random_integer(size=1 , len(parents))
                o1 ← join first part of P1 and second part of P2
                o2 ← join first part of P2 and second part of P1
                Offspring U o1 U o2
                If(Offspringt in Po) do
                  Offspringt← mutation(Offspringt)
        End
        Return offspring
```

3. Mutation

A mutation is a process used to create genetic diversity in newly generated offspring. For our problem mutation is only called when newly generated offspring are already existing in the current population. In that case, the bit-flip mutation is done. In this type of mutation, one or more random bits are selected and flipped. For example, if the randomly selected bit is 0 then it is flipped to 1 and vice versa. Algorithm 6 is used in our proposed tool for mutating individuals.

Algorithm 6: Mutation

```
 Input: Po , Offspring
Output: Offspring
Procedure:
        t ← 0
        Offspring← $\varphi$
        While(t < length(Offsping)) do:
                while (offspring(t) in Po) do:
                   ln ←length(offspring(t))
                   point←random_integer(start:0, stop:ln)
                   offspring(t) ← flip bit(offspring(t) , point)
                 end
        End
        Return Offspring
```

After unique offspring are generated, they are replaced with the selected parents using the index values in the array (indx). This updated population is again processed with a genetic algorithm unless the fitness value of the entire population reaches the threshold, or the number of generations is completed. Once the genetic algorithm is terminated it outputs the current updated population with its population fitness value along with the fitness values of each member present in the population.

3.2 Experimentation and Results

In this section, the performance of proposed module for s/w approximation (data compression) is being evaluated in python. The proposed genetic algorithm is implemented using Python 3.8 and cuda (11.1) for NIVIDA GPU. We performed experiments with different user requirements of accuracy loss, percentage of memory being saved along with preference variable (**α** and **β**) and threshold values. The algorithm is tested for models and datasets defined in section 3.2.1.

3.2.1 Models and Datasets

The proposed algorithm is tested for same models used for data collection process in section 3.1.2 presented in Table 3. In this section we will the discuss the architecture of these models and the attributes of datasets in detail.

3.2.1.1 *LeNet-5 Network*

Lenet-5 is a simple and straightforward pre-trained models suggested by Yann LeCun [30] in 1998 for multiclass classification. The architecture designed was to recognize handwritten and machine printed characters. Lenet-5 consists of 5 layers with trainable parameters. It has three convolution layers with a sequence of average pooling layers in between. At last, it contains two fully connected layers (dense). It uses SoftMax classifier to classify the input images to their respective classes. The architecture of LeNet-5 can be seen in Figure 13. For our data collection and optimization process we trained LeNet-5 on MNIST dataset.

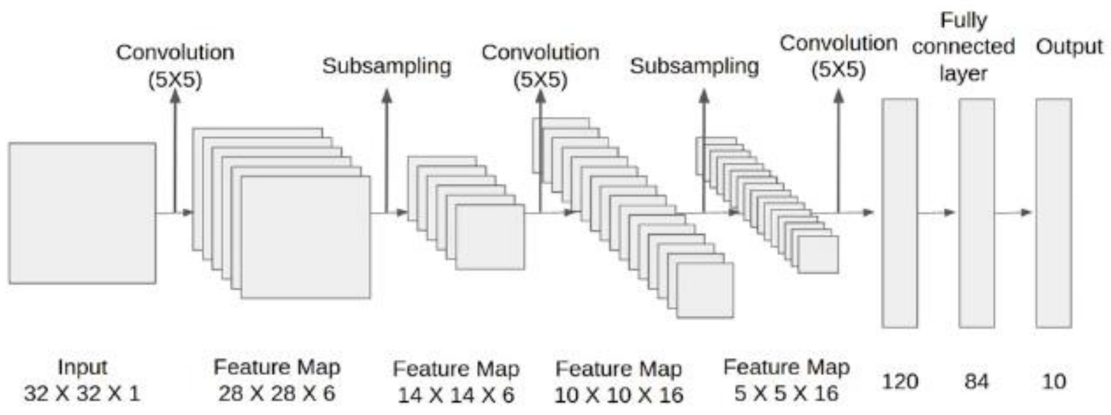


Figure 13: Architecture of LeNet-5 [30]

3.2.1.2 *MNIST Dataset*

The MNIST dataset (Modified National Institute of Standards and Technology database) is a collection of handwritten digits. It is a subset of NIST Special Database 3 and Special Database 1 [31]. The images for MNIST contains a training set of 60,000 images, and a test dataset of 10,000 images of 28x28 pixels.

3.2.1.3 *Cifar-Quick Network*

Cifar-Quick is an fast learning model designed for the fast classification of Cifar-10 dataset developed by Yiren Zhou [32]. Cifar-Quick consists of 4 layers with trainable parameters. It has three convolution layers with a sequence of maximum pooling layers in between. At last, it contains single fully connected layers (dense). It uses SoftMax classifier to classify the input images to their respective classes. The architecture of Cifar-Quick can be seen in Figure 14.

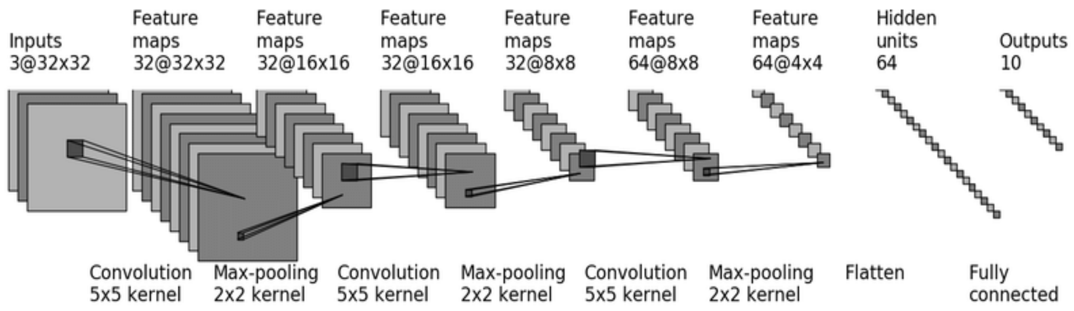


Figure 14: Architecture of Cifar-Quick [32]

3.2.1.4 *Cifar-10 Dataset*

The CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset. They were collected by Alex Krizhevsky. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images [33].

3.2.1.5 *VGG-16 Network*

The VGG16 Architecture is developed by Karen Simonyan 2014 [34]. This architecture has 16 layers. 13 convolutional layers with 5 max-pooling layers adjusted in between leading to 3 fully connected layer and single output layer using SoftMax for activation for multiclass classification. VGG16 architecture is an improvement version for Alex Net [36] replacing the large kernel-sized filters with multiple 3 x 3 kernel-sized filters one after the other. The architecture diagram of VGG-16 is presented in Figure 15.

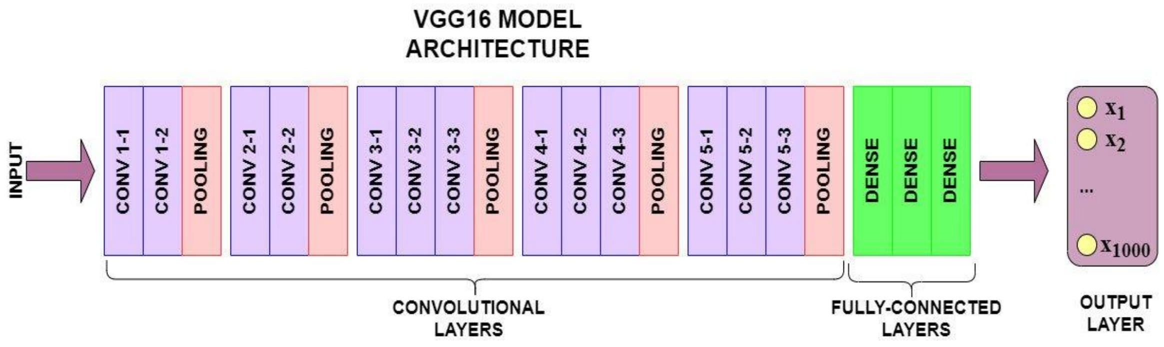


Figure 15: Architecture of VGG-16 [35]

For our experimentation we used this model to classify ImageNet Large Scale Visual Recognition Challenge dataset (ILSVR) 2012 dataset

3.2.1.6 *ImageNet Dataset 2012*

For our work we utilized a most highly used subset of ImageNet is the ILSVRC 2012-2017 image classification and localization dataset. This dataset spans 10 object classes and contains 13394 total images, 9469 training images, 3925 validation images. This subset dataset is available at Kaggle at [37].

3.2.2 Performance Evaluation

To determine the performance of our proposed optimizer we estimated the “N-levels” for quantization using following three cases shown in Table 5 for different and same levels at Dense and Convolution layers for all quantization types (Q-types) mentioned in section 3.1.1.1

Table 5: Cases for testing proposed optimizer tool

Parameters	Case 1	Case 2	Case 3
Iterations	100	200	50
Size	10	10	10
Crossovers	2	2	3
User error	0.01	0.1	0.05
User mem	0.4	0.35	0.50
Alpha	0.6	0.6	0.5
Beta	0.4	0.4	0.5
Threshold	0.05	0.1	0.07

The genetic algorithm was tested on the dataset and network described in section 3.2.1 for all Q-types and scenarios

3.2.2.1 *Experimentation Results for LeNet-5*

The following Table 6 and 7 shows the results of outputted fitness values and population of best “N-levels” bases on above three cases.

Table 6: The Output of optimizer for Same N levels tested for LeNet-5

Same Levels for both Dense and Convolutional layers									
Q-type	Case 1			Case 2			Case 3		
	iter	MSE	Final Pop	iter	MSE	Final Pop	iter	MSE	Final Pop
Uniform	100	0.075462	2216, 246, 151, 5, 2152, 246, 31, 146, 145, 168	4	0.092879	803, 246, 151, 5, 250, 246, 1315, 146, 145, 250	50	0.090953	2324, 246, 151, 5, 4, 246, 21, 146, 145, 2838
Non-Uniform	100	0.0823821	169, 3221, 3204, 178, 17, 126, 55, 81, 124, 95	14	0.0954563	169, 21, 3379, 178, 17, 9, 55, 81, 124, 3507	50	0.0969911	169, 2645, 178, 178, 17, 52, 55, 81, 124, 2645
Asymmetric	100	0.0728364	97, 49, 2108, 73, 90, 3882, 115, 4012, 55, 2568	1	0.0947414	97, 253, 129, 73, 90, 129, 115, 288, 181, 123	50	0.0923694	2798, 61, 50, 73, 90, 26, 2810, 53, 4053, 4012

Table 7: The Output of optimizer for Different N levels tested for LeNet-5

Different Levels for both Dense and Convolutional layers									
Q-type	Case 1			Case 2			Case 3		
	iter	MSE	Final Pop	iter	MSE	Final Pop	iter	MSE	Final Pop
Uniform	100	0.08564331	[129, 32], [194, 69], [1070, 2787], [191, 74], [324, 91], [97, 68], [1078, 2763], [55, 90], [203, 48], [203, 70]	4	0.09987660	[129, 32], [194, 69], [229, 122], [191, 74], [2223, 1711], [97, 68], [2285, 1791], [55, 90], [203, 48], [203, 70]	50	0.11088225	[129, 32], [194, 69], [229, 122], [191, 74], [3391, 949], [97, 68], [2363, 405], [55, 90], [203, 48], [203, 70]
Non-Uniform	100	0.09771716	[182, 109], [3570, 3960], [3516, 497], [2352, 4000], [3504, 503], [3832, 3944], [279, 27], [2288, 4064], [255, 85], [3448, 3963]	9	0.09887903	[182, 109], [399, 1321], [200, 133], [397, 32], [69, 171], [2189, 1377], [279, 27], [299, 147], [255, 85], [216, 151]	50	0.10147556	[182, 109], [2305, 3987], [200, 133], [2457, 4019], [69, 171], [2890, 3847], [279, 27], [1143, 2036], [255, 85], [1783, 1520]
Asymmetric	100	0.07824536	[1966, 2734], [95, 23], [295, 56], [655, 118], [900, 65], [2803, 4059], [1963, 2238], [156, 47], [156, 152], [102, 207]	4	0.09363832	[41, 236], [95, 23], [295, 56], [311, 359], [259, 1319], [116, 276], [144, 235], [156, 47], [156, 152], [102, 207]	50	0.09951856	[501, 97], [95, 23], [295, 56], [73, 1174], [1833, 3941], [422, 24], [3567, 3826], [156, 47], [156, 152], [2120, 3206]

For the verification of our results, we used the outputted population of N-levels provided for both cases to quantize the actual LeNet model weights and calculated the accuracy loss at CNN inference and percentage of saved memory. Once these values were obtained, we determined the mean square error with respect to the user requirements inputted to GA. The following graphs were obtained plotting the absolute error of the MSE calculate using GA and MSE calculate using actual Quantized CNN model in Figure 16(a) and (b)

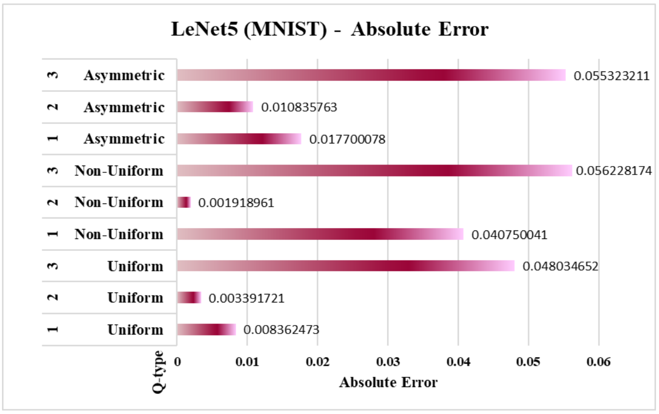


Figure 16(a): Different levels for Dense and Convolutional layers

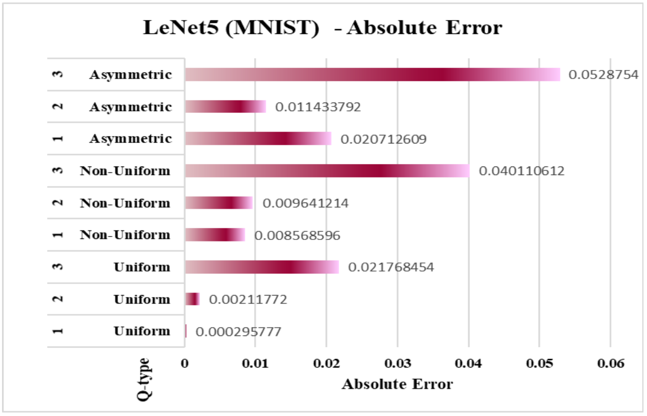


Figure 16(b): Same levels for Dense and Convolutional layers

3.2.2.2 *Experimentation Results for Cifar-Quick*

Cifar-Quick was trained for Cifar10 dataset. The Table 8 and 9 shows the results of outputted fitness values and population of best "N-levels" bases on above three cases in Table 5.

Table 8: The Output of optimizer for Same N levels tested for Cifar-Quick

Same Levels for both Dense and Convolutional layers									
Q-type	**Case 1**			**Case 2**			**Case 3**		
	iter	**MSE**	**Final Pop**	**iter**	**MSE**	**Final Pop**	**iter**	**MSE**	**Final Pop**
Uniform	100	0.079987	86, 118, 151, 5, 705, 3009, 49, 146, 145, 235	2	0.0959615	1315, 246, 151, 5, 250, 246, 35, 146, 145, 250	50	0.0924240	59, 114, 87, 5, 13, 2525, 35, 3029, 145, 107
Non-Uniform	100	0.0737016	169, 2846, 178, 178, 17, 2846, 55, 81, 124, 215	4	0.0939973	169, 2312, 178, 179, 17, 265, 55, 81, 124, 215	50	0.0988072	169, 137, 263, 178, 17, 8, 55, 81, 124, 2309
Asymmetric	100	0.0731642	97, 3839, 4052, 73, 90, 1611, 115, 68, 61, 1611	2	0.0951072	97, 253, 129, 73, 90, 129, 115, 288, 181, 123	50	0.0846333	97, 3905, 76, 73, 90, 103, 115, 1626, 34, 1626

Table 9: The Output of optimizer for Different N levels tested for Cifar-Quick

Different Levels for both Dense and Convolutional layers									
Q-type	Case 1			Case 2			Case 3		
	iter	MSE	Final Pop	iter	MSE	Final Pop	iter	MSE	Final Pop
Uniform	100	0.07862502	[129, 32], [194, 69], [229, 122], [191, 74], [3119, 1246], [97, 68], [3151, 1246], [55, 90], [203, 48], [203, 70]	5	0.09574368	[129, 32], [194, 69], [229, 122], [191, 74], [2285, 823], [97, 68], [2125, 827], [55, 90], [203, 48], [203, 70]	50	0.09167238	[129, 32], [194, 69], [229, 122], [191, 74], [2188, 2233], [97, 68], [2221, 2749], [55, 90], [203, 48], [203, 70]
Non-Uniform	100	0.10155447	[2845, 3762], [3664, 3757], [2846, 1682], [3589, 3860], [3489, 3873], [1404, 4060], [279, 27], [3841, 3496], [2465, 3944], [2441, 3775]	6	0.09894566	[182, 109], [265, 32], [200, 133], [261, 32], [69, 171], [320, 102], [279, 27], [449, 396], [255, 85], [897, 910]	50	0.10442580	[182, 109], [2806, 4068], [200, 133], [1596, 3952], [1849, 1719], [2857, 4093], [279, 27], [3917, 3632], [255, 85], [1837, 1713]
Asymmetric	100	0.08072956	[2298, 4060], [95, 23], [295, 56], [35, 2054], [803, 2318], [3417, 3948], [254, 4086], [156, 47], [156, 152], [3583, 3692]	2	0.09141822	[41, 236], [95, 23], [295, 56], [122, 294], [295, 295], [116, 276], [144, 235], [156, 47], [156, 152], [102, 207]	50	0.10710180	[2096, 4014], [1406, 1697], [3383, 3697], [2931, 4015], [1981, 3824], [3696, 4012], [2420, 3903], [3190, 3825], [2663, 3694], [1910, 1969]

To verify the results of Cifar-Quick same procedure is followed as stated in pervious section. The Graph in Figure 17(a) and (b) shows the absolute error of actual computed MSE and MSE computed through GA.

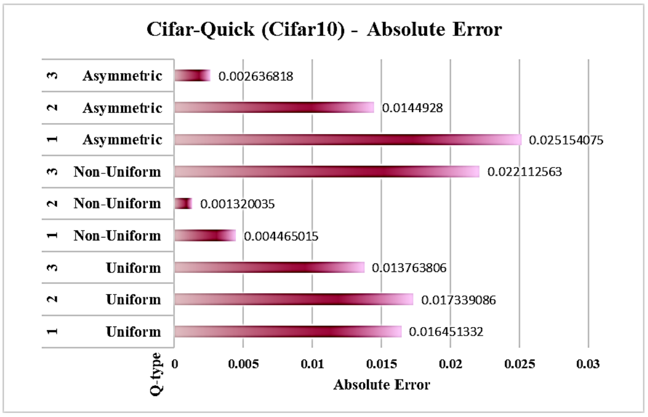


Figure 17(a): Different levels for Dense and Convolutional layers

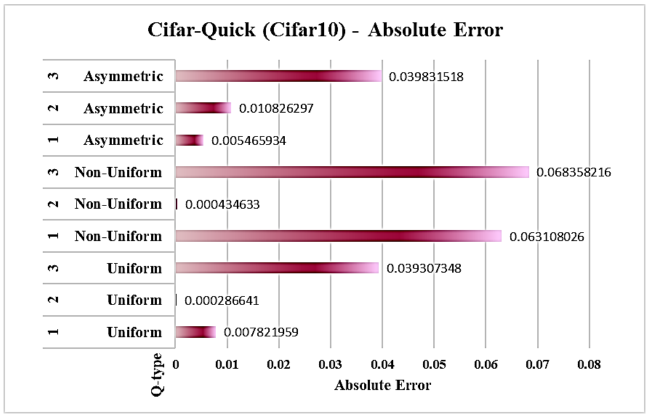


Figure 17(b): Same levels for Dense and Convolutional layers

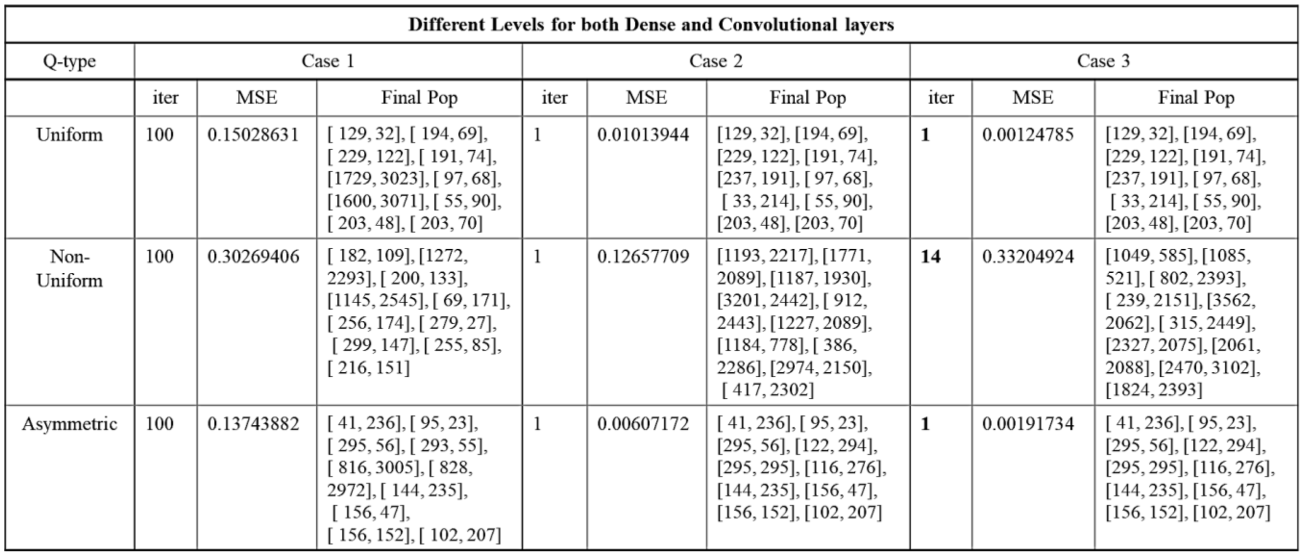
3.2.2.3 *Experimentation Results for VGG-16*

VGG16 was trained for ImageNet dataset. The following Tables 10 and 11 shows the results of outputted fitness values and population of best "N-levels" bases on above three cases in Table 5.

Table 10: The Output of optimizer for Same N levels tested for VGG-16

Same Levels for both Dense and Convolutional layers									
Q-type	**Case 1**			**Case 2**			**Case 3**		
	iter	**MSE**	**Final Pop**	**iter**	**MSE**	**Final Pop**	**iter**	**MSE**	**Final Pop**
Uniform	100	0.079987	86, 118, 151, 5, 705, 3009, 49, 146, 145, 235	2	0.0959615	1315, 246, 151, 5, 250, 246, 35, 146, 145, 250	50	0.0924240	59, 114, 87, 5, 13, 2525, 35, 3029, 145, 107
Non-Uniform	100	0.1452532	169, 3028, 178, 178, 17, 39, 55, 81, 124, 2774	2	0.0079493	169, 263, 178, 175, 17, 236, 55, 81, 124, 215	1	0.0014192	169, 263, 178, 178, 17, 236, 55, 81, 124, 215
Asymmetric	100	0.3192632	97, 2214, 129, 73, 90, 129, 115, 2214, 181, 123	200	0.1265083	2225, 2480, 2287, 1455, 672, 1008, 1123, 1409, 1696, 2163	50	0.2783122	2981, 1402, 2233, 2271, 2138, 2133, 2163, 127, 255, 2193

Table 11: The Output of optimizer for Different N levels tested for VGG-16



Different Levels for both Dense and Convolutional layers									
Q-type	Case 1			Case 2			Case 3		
	iter	MSE	Final Pop	iter	MSE	Final Pop	iter	MSE	Final Pop
Uniform	100	0.15028631	[129, 32], [194, 69], [229, 122], [191, 74], [1729, 3023], [97, 68], [1600, 3071], [55, 90], [203, 48], [203, 70]	1	0.01013944	[129, 32], [194, 69], [229, 122], [191, 74], [237, 191], [97, 68], [33, 214], [55, 90], [203, 48], [203, 70]	**1**	0.00124785	[129, 32], [194, 69], [229, 122], [191, 74], [237, 191], [97, 68], [33, 214], [55, 90], [203, 48], [203, 70]
Non-Uniform	100	0.30269406	[182, 109], [1272, 2293], [200, 133], [1145, 2545], [69, 171], [256, 174], [279, 27], [299, 147], [255, 85], [216, 151]	1	0.12657709	[1193, 2217], [1771, 2089], [1187, 1930], [3201, 2442], [912, 2443], [1227, 2089], [1184, 778], [386, 2286], [2974, 2150], [417, 2302]	**14**	0.33204924	[1049, 585], [1085, 521], [802, 2393], [239, 2151], [3562, 2062], [315, 2449], [2327, 2075], [2061, 2088], [2470, 3102], [1824, 2393]
Asymmetric	100	0.13743882	[41, 236], [95, 23], [295, 56], [293, 55], [816, 3005], [828, 2972], [144, 235], [156, 47], [156, 152], [102, 207]	1	0.00607172	[41, 236], [95, 23], [295, 56], [122, 294], [295, 295], [116, 276], [144, 235], [156, 47], [156, 152], [102, 207]	**1**	0.00191734	[41, 236], [95, 23], [295, 56], [122, 294], [295, 295], [116, 276], [144, 235], [156, 47], [156, 152], [102, 207]

The following graphs in Figure 18(a) and (b) shows the absolute error of actual MSE and MSE from GA.

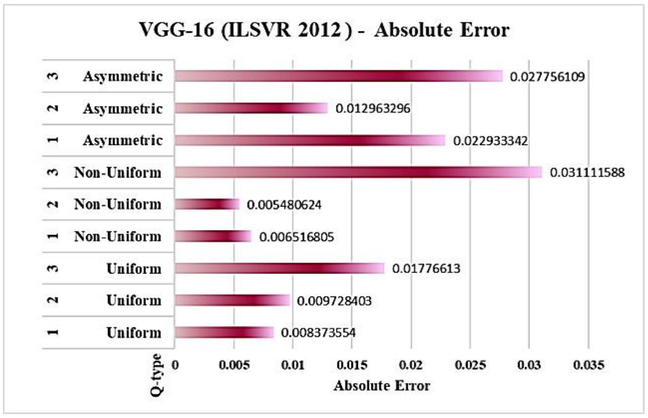


Figure 18(a): Different levels for Dense and Convolutional layers

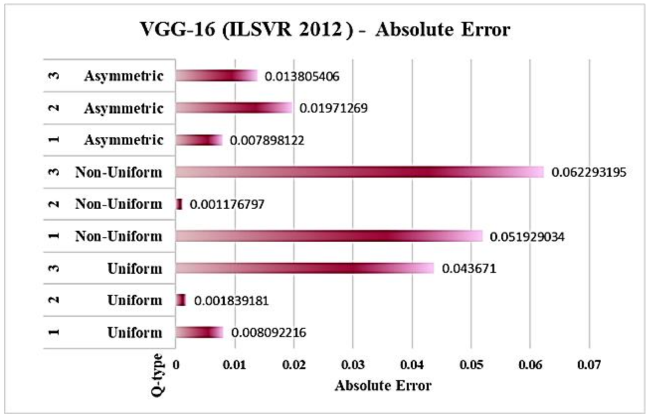


Figure 18(b): Same levels for Dense and Convolutional layers

3.2.3 Final Discussion

From section 3.2.2.1 – 3.2.2.3 we can see that the different of highest error between both MSEs is of 0.0683 in case of CIFAR-Quick for non-uniform Q-type. To determine for which Q-type the highest error occurred we plotted the average of all absolute errors with respect to their Q-types in Figure 19.

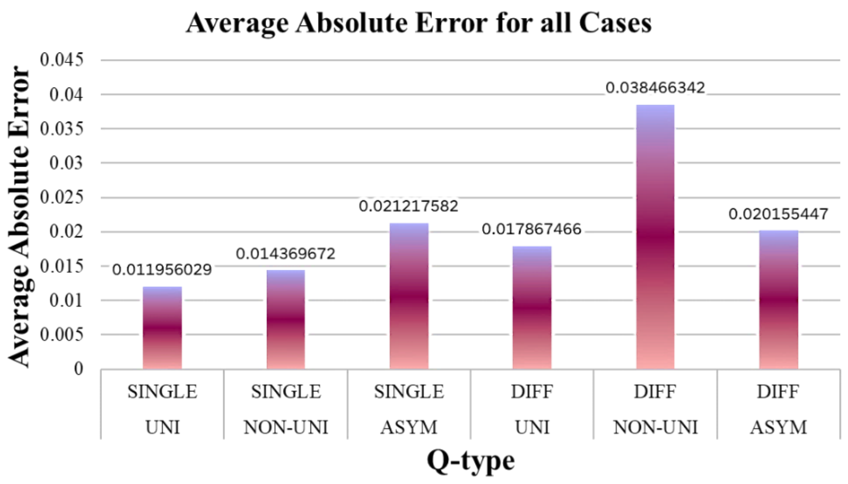


Figure 19: Average Absolute Error for all cases w.r.t Q-types

The above graph shows that the highest average error i.e., 0.038466342 given by non-uniform quantization for different levels in convolutional and dense layers. This average error is very minimal and could be ignored. But if we are to enhance our regression equations, these errors can be further reduced. The regression equation only considers and parameters on a CNN model and the number of convolutional layers used in CNN model ignoring all information regarding dataset to classified with a CNN model. If a fitness function is designed to add this information, there is chance to reduce the error in our proposed design.

3.3 Chapter Summary

This chapter is divided into two parts the first part discusses the methodology used for building the proposed tool. The second part discuss the experimentation done for the testing of the proposed module. The proposed tool is based on GA estimating best quantization levels with respect to user requirements. The fitness function of proposed module utilizes regression equations determines the relationship of quantization levels with accuracy loss and memory reduction. The proposed module was tested for datasets and networks mentioned in section 3.2.1 for three different types of quantization mentioned in section 3.1.1. The proposed module is time efficient and returned the optimal parameters efficiently with respect to user requirements based on mean squared error. The estimated parameters were then used to quantize actual weights of CNN model and mean squared error with user requirements was obtained. The absolute error between the originally computed MSE and MSE computed using GA was determined for each quantization type. The highest average absolute error calculated between two MSEs is of 0.038.

Chapter-4: Real-Time Hardware decoder for data decompression in CNNs

The proposed de-compression decoder design is based on the Canonical Huffman Coding, which is a subset of the Huffman Coding having numerical sequence property. The canonized compression of weights is beneficial for our decoder. In basic Huffman, the coding decoder must pass through an entire Huffman tree for decoding an encoded symbol resulting in the utilization of several LUTs increasing memory of the decoder. In contrast, the canonical Huffman decoder only requires a number of bits to generate decoded symbols.

4.1 The Proposed Design

To understand the overall framework of our proposed decoding mechanism let us consider a maximum 8-bit code length. Our process starts with taking an 8-bit input from a sequentially sliding input stream. A best way to decode a data is in a serial manner or bit by bit, However, this is not favorable choice as it may take more than one clock cycle to decode the data. The first goal is to avoid this limitation in order to do so data is processed using an 8-input multiplexer in rising bit order. The next goal is to verify the encoded symbol for this purpose a single LUT is used. As mentioned before, in canonical Huffman, decoder only requires bit information to decode the encoded data. Therefore, we have used a single LUT containing bit information of each encoded symbol.Once the code is verified, it is used to calculate the address of the memory containing decoded symbols. Meanwhile the input is updated accordingly. The process is carried out till all weights are decoded in the encoded data. Figure 20 provides the overall architecture of our proposed design. The decoder contains two main sub-modules. Check_valid used to test the validity of the data and address_generator to calculate the memory address of decoded symbol. The hash tables are used for storing the decoded weights.

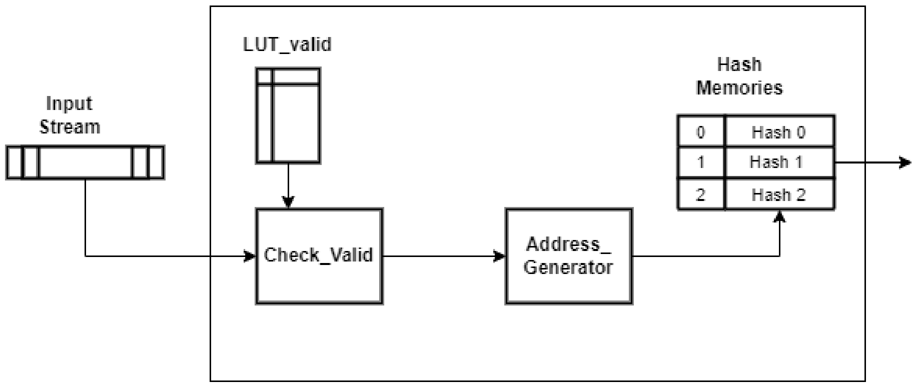


Figure 20 The Overall Architecture of proposed Module

4.1.1 LUT valid

LUT_valid is a simple Look-Up Table (LUT) consisting of two columns one representing the address of LUT and the second column representing the number of bits for each codeword. LUT_valid inputs codeword as an address/index, implying that each codeword is unique doesn't repeat. In canonical Huffman encoding, it is very unlikely for a codeword to repeat with different bit lengths. An example of LUT_valid is represented in Table 12.

Table 12: The representation of valid LUT

Code / Index	Number of bits (length)
0000	3
0001	3
0010	3
0011	0
0100	0
0101	0
0110	4
0111	4
1000	4
1001	4
1010	4
1011	4
1100	4
1101	4
1110	0
1111	0

Length zero in LUT_valid represents that code is invalid and doesn't exist.

4.1.2 Check Valid

Check valid determines the validity of input code using simplified LUT_valid. The internal Framework of check_valid module can be seen in Figure 21.

First Input-Stream of 128bits is loaded from the encoded memory. Each stream in encoded memory is set in a way that all input streams contain maximum N symbols i.e., 24 maximum codes in our case. At initial clock cycle MUX 0 updates the IR register with input-stream controlled by sel_0 signal. Control signal sel_0 is initially set to 0 and then it is set to 1 to take input from MUX3.

At the Nth-1 state it is set to 0 so that at next coming IR can be updated with new input-stream without any stall cycles.

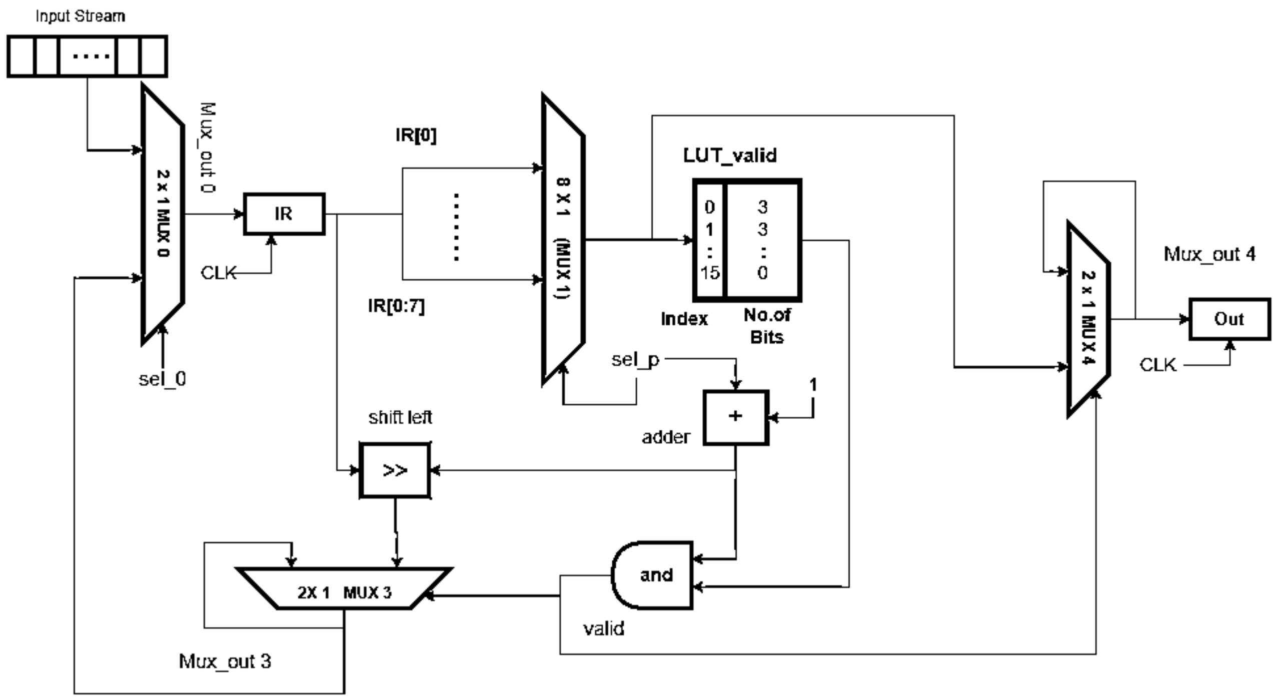


Figure 21: The Internal Structure of Check Valid Module

IR register provides the input to MUX 1 which is an 8x1 multiplexer controlled by sel_p. Mux 1 takes first 8-bits saved in IR as input. Taking its lowest single bit as first input and increasing the bit size by 1 for its other. For example, MUX 1 first input will be IR [0] containing single bit and incrementing the bit size for each input, its last input will be IR [0:7] containing 8 bits. The output of MUX 1 is treated as memory address to LUT_valid. The data value at that address is noted and send to the input of AND gate to compare it with the output of sel_p + 1. If the inputs at AND are same, it will output 1 else 0. The output of AND controls MUX 3 and MUX 4.

Simultaneously IR register is shifted left using sel_p+1 connected to one of MUX 3 inputs. Once the input gets verified by AND gate it enables control signal of MUX 3 and MUX 4 to 1. MUX 3 outputs the shifted IR by sel_p+1 updating in IR for next clock cycle. While MUX 4 outputs the output provided by MUX1. MUX 4 output is updated to out register with positive clock cycle. This output is also directly connected to address generator to neglect latency cycles. This process is repeated for all inputs of MUX 1 with a single clock cycle. It is to be noted that AND gate, Adder and Shift-Left module are working with respect to sel_p.

4.1.3 Address Generator

Address Generator uses the codeword input to compute the address of the hash table and hash index. These hash tables contains the decompressed weights sorted using the hash function during compression. For determining the address of the hash table and hash table index codeword is used as a key in the hash function. We have used single and multiple hash tables for our work. For single hash function only memory address is to be generated. While for multiple hash tables, table address is also required with memory address as shown in Figure 22(a) and (b).

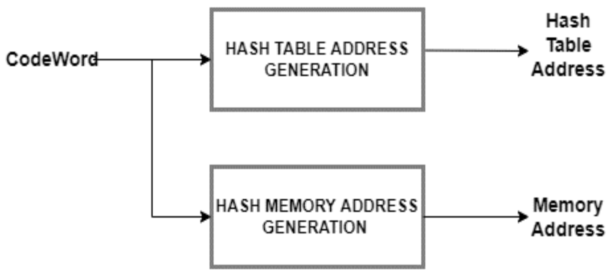


Figure 22(a): Multiple Hash Tables

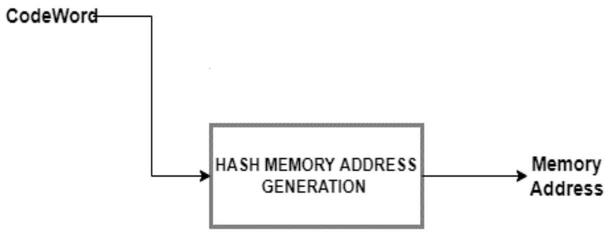


Figure 22(b): Single Hash Table

4.1.3.1 HASH Functions

Hash functions eliminates the need of searching dictionary which is time taking. Hash functions are defined differently with respect to number of elements to be stored in them with respect to number memory locations available. Generally, the number of memory locations selected using following equation 16.

$$\text{Needed Locations} = f \times 2.6 \tag{16}$$

Where f represents the number of unique codes or symbols present in coded file. For our experimentation, we determined the equations for hash function for two cases for a number of symbols (N) 32 and 48 respectively. Hash equations are determined in such way that no memory location of a symbol overlaps with memory location of another symbol. To estimate the Hash equation efficiently following parameters are used:

- **M:** No. of Memory Locations in a single Hash Table
- **T:** No. of Hash Tables
- **f:** No. of unique codes
- **P:** randomly generated prime number

The hash equations are determined for multiple and single hash tables for both cases are represented in Table 13.

Table 13: The Equation determined for hash function for the efficiently placement of decompressed weight

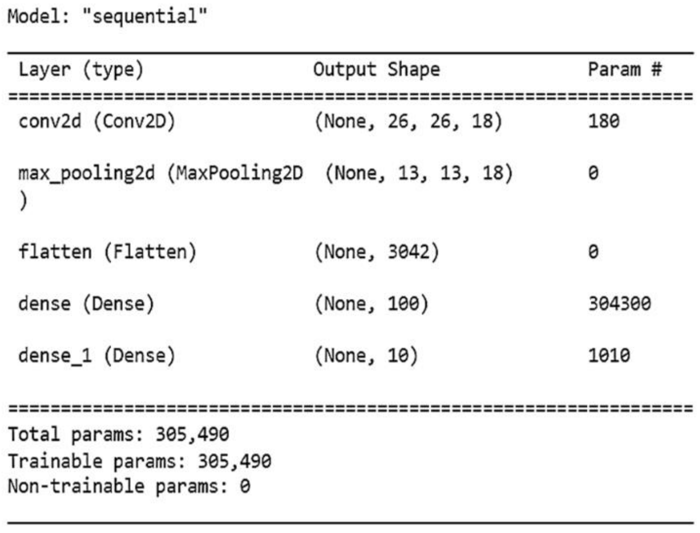
Hash Equations		
Case	***Equation For Single Hash Table***	
N= 32	$\text{Hash Mem}_{\text{adr}} = \text{Codeword} \ \% \ \text{M}$	(17)
N = 48	$\text{Hash Mem}_{\text{adr}} = \text{Coderword} \ \% \ (\text{f}/2) \% \ \text{M}$	(18)
	Equation For Multi Hash Table	
N = 32	$\text{Hash Table}_{\text{adr}} = (\text{Hash Mem}_{\text{adr}} + \text{p} \times \text{bit}_{\text{size}}) \ \% \ \text{T}$	(19)
N = 48	$\text{Hash Table}_{\text{adr}} = (\text{Hash Mem}_{\text{adr}} \times \text{bit}_{\text{size}} \ \% \ \text{p}) \ \% \ \text{T}$	(20)

4.2 Experiments and Results

This section analyzes the performance of our proposed architecture. We implemented our architecture using HDL Verilog in Xilinx Inc 14.7 with the target device set as XC7A200T of family Aritx7. 2Kb memory space is required for storing our LUT (LUT_valid) as it contains 256 memory locations, to store bit information of maximum 8bit symbols. We experimented with our design using single and multiple hash memories.

4.2.1 Dataset and CNN Model

To verify its efficiency in decoding we tested our decoder for decoding weights of a simple CNN model designed in python 3.8 using TensorFlow library. The model consists of single convolutional layer with 28 filters of size 3x3 followed by a maximum pooling layer. It contains one fully connected dense layer with 100 nodes and one output layer with SoftMax activation for the testing of MNIST dataset [31]. The said model achieved 97.8% accuracy with 97MB size of its weights. The structure of CNN model is shown in Figure 23.


```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 18)        180

 max_pooling2d (MaxPooling2D  (None, 13, 13, 18)       0
 )

 flatten (Flatten)           (None, 3042)              0

 dense (Dense)               (None, 100)               304300

 dense_1 (Dense)             (None, 10)                1010

=================================================================
Total params: 305,490
Trainable params: 305,490
Non-trainable params: 0
_________________________________________________________________
```

Figure 23: CNN Model Defined for testing our decoder

For testing our decoder, CNN weights were quantized, to 16-bits with N=32 and N=48 levels with minimal loss in accuracy. Table 14 shows the accuracy compression of quantized weights with original weights and memory size of original weights with encoded weights.

Table 14:Accuracy and Memory Comparison of Original and Encoded CNN weights

CNN Model	Accuracy	Memory
Original Model	97.87 %	9.7 MB
Quantized Model (N = 32)	97.74%	0.96 MB
Quantized Model (N = 48)	97.77%	0.97 MB

These quantized weights are passed to Canonical Huffman encoder, designed in python to generate a Lookup table stored in LUT_valid. Then using the Hash equations mentioned in Table 13 we updated Hash memories. For decompression, encoded weights were stored in single-dimensional arrays of 128 bits passed to the input stream of our proposed decoder once all 128 bits are processed the next input is updated without having a stall cycle.

4.2.2 Memory Requirements for Single and Multiple Hash Tables

We calculated the memory requirements for both cases for using single and multiple hash tables containing memory locations of 32-bit. The required memory for both cases can be seen in Table 15.

Table 15: Comparison of Hash Table Memory for Single and Multiple Tables

Memory Requirement			
Case	***T***	***M***	***Req Storage***
N: 32	1	128	4KB
	3	16	1KB
N: 48	1	128	4KB
	5	16	2KB

For saving the decompressed weights at unique memory locations we require 128 memory locations of 32 bit for single hash table. While only 16 memory locations per table are required in case of multiple hash tables i.e., total 48 memory location for case N=32 and total of 80 memory location for case N=48. Hence, we can say that using small chunk memories instead of using a single chunk can decrease our memory requirement.

4.2.3 Resource Utilization

Resource utilization for both scenarios were noted for target device XC7A200T-3fbg484 of family Aritix 7 in Xilinx. Inc 14.7 are shown in Table 16.

Table 16: Resource Utilization for Proposed Framework

Resource Utilization				
Device: XC7A200T	***Slice Reg***	***Slice LUT***	***LUT BFFs***	***IOBs***
N = 32	415	481	490	158
N = 48	410	456	467	158

Our decoder only uses 1% of the system slice registers providing us efficient results with decoding a single weight within one clock cycle with maximum frequency of 408.97MHz. The latency of system is determined using number of latent cycles into maximum clock period i.e., 2.445ns, 408MHz.

4.2.4 Xilinx Simulation Results

Results for both cases were simulated on Xilinx. The encoded weight files were stored in an 2D array each row of that array is of 128bits of data containing 24 symbols. Hence after decoding 23 symbols of data perfectly the signal is sent to MUX 0 to update the IR with next input stream present in the weight array. So, when 24th clock cycle occurs the IR gets updated without creating any hindrance in the system performance.

Xilinx Simulation results for both cases are shown Figure 24(a) and 24(b). Proposed Framework efficiently decodes a single element within one clock cycle.

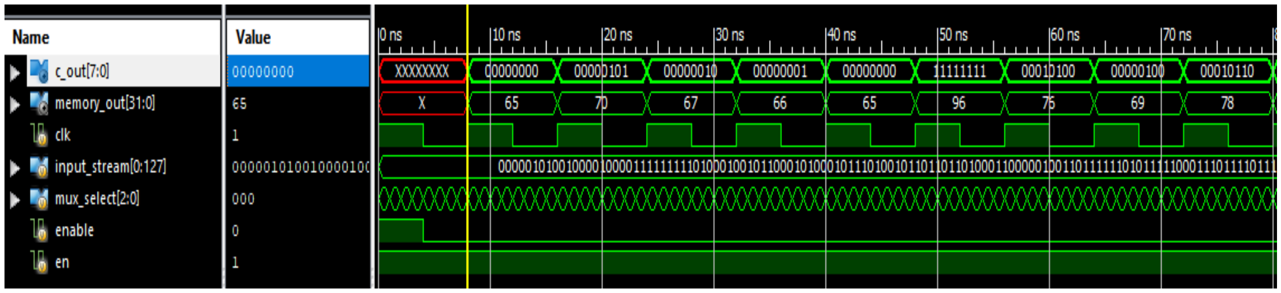


Figure 24(a): Xilinx Simulation of proposed decoder, decoding CNN weight for case N = 32

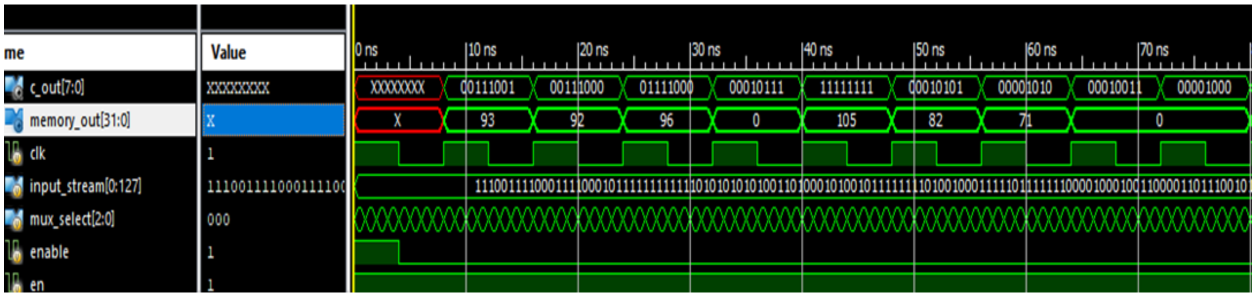


Figure 24(b): Xilinx Simulation of proposed decoder, decoding CNN weight for case N = 48

4.2.5 Comparison with Literature

It is noted that the proposed design can efficiently decode a single weight within single clock cycle. It only requires single LUT instead of multiple LUTs used in literature. It lessens the memory storage required to store multiple LUTs. Multiple hash memories proved effective in reducing memory requirements and access time. We compared our proposed work with pervious decoders and concluded that our decoder is much simpler and easier to implement and requires less amount of system resources as shown in Table 17.

Table 17: Comparison with Resource Utilization and Maximum Frequency

Resource Utilization Comparison				
Decoders	***Device***	***Slice Reg***	***Slice LUT***	***Max. Clock Frequency***
Proposed	XC7A200T	415	481	408.97MHz
	Virtex 5	408	424	270.033MHz
	XC7K325T	414	614	409.10MHz
Canonical Decoder	XC7K325T	8044	N/A	50 MHz
Huffman Decoder	Virtex 5	179	N/A	263.2 MHz
Parallel Huffman Decoder	Altera RC240-3	N/A	1145	11.54 MHz
	Alter RC240-4	N/A	1145	9.91 MHz

4.3 Chapter Summary

We designed an efficient Canonical Huffman decoder in Xilinx 14.7 using XC7A200T device to decode 8-bit symbols. This decoder can be integrated to machine learning algorithms implemented to embedded devices. The proposed framework efficiently decodes a sisngle weight within one clock cycle using 1% of system resources. For our experiment we assumed variable length code with highest 8-bit length to implement our design. We only had to store N=32 and N= 48 memory locations in LUT valid. But in cases where we have a lot more symbols and symbols with bit length greater than 8-bit it can enhance the memory requirements to store our LUT valid. Hence optimization strategies can be developed to further enhance our lookup table, so it does not get problematic to store in memory if it contains symbols more than 8-bits.

Chapter-5: Automated tool for the selection of best approximate units (multipliers) in CNNs

5.1 Proposed Automated Tool

The proposed Framework is specifically designed for utilizing approximate multipliers in CNNs. For this purpose, we used an open-source tf-approximate library which is an extension of TensorFlow in python. This library enables approximate computations at Convolutional layers of any defined CNN model. This library does not allow approximate computations at Dense layers. According to research, convolutional layers are more computationally extensive in comparison to dense layers hence the approximate computation is preferred only at convolutional layers. The proposed framework for this module is also based on GA and is shown in Figure 25. The genetic algorithm requires a set of approximate multipliers, tf-approximate library, and user requirements. Hence, before jumping into the internal working of GA let's discuss the approximate multipliers used for our problem and tf-approximate library.

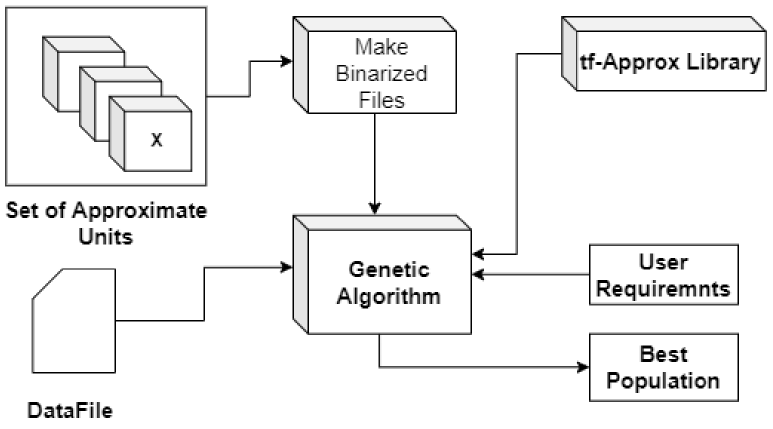
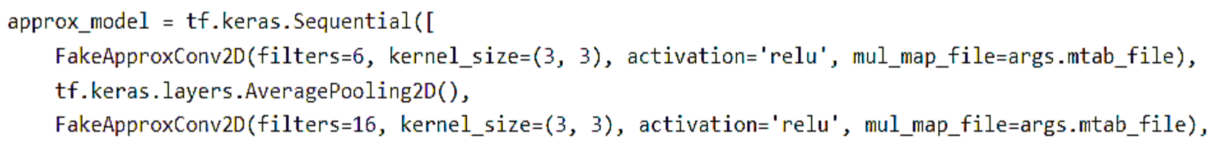


Figure 25: The overall framework of proposed algorithm

5.1.1 tf-approximate library

The tf-approximate library extends the TensorFlow library and is provided by V.Mrazek [36][37]. It allows approximate computation in Convolutional layers using FakeApproxConv2D. This layer is implemented with reduced precision (usually 8 bits) using approximate units (multipliers). The approximate convolutional layer inputs an approximate multiplier defined by a binary LUT as a parameter. Hence for the approximate CNN inference, simply switch the original Conv2D by the FakeApproxConv2D to perform the approximate calculation at convolutional layers as shown in Figure 26.



```
approx_model = tf.keras.Sequential([
    FakeApproxConv2D(filters=6, kernel_size=(3, 3), activation='relu', mul_map_file=args.mtab_file),
    tf.keras.layers.AveragePooling2D(),
    FakeApproxConv2D(filters=16, kernel_size=(3, 3), activation='relu', mul_map_file=args.mtab_file),
```

Figure 26: The implementation of approximate layer in python keras

In above mul_map_file is an argument which inputs binary file of specified multiplier to be used in approximation. The said library is available at [36][37]. This library requires following prerequisites to build the library

1. CUDA SDK (10.0+)
2. TensorFlow (2.2.0-)
3. Ubuntu/Debian System

In our framework this library is used in fitness function for the estimation of accuracy loss.

5.1.2 Approximate Multipliers

The approximate units used in this research are EvoApprox8b provided by V.Mrazek [38]. This approximate library contains a total 500 dominated 8bit signed approximate multipliers. The EvoApprox8b library provides Verilog, MATLAB and C models of all approximate circuits. The developers of these multipliers have tested them using Cadence Encounter RTL Compiler and TSMC 180 nm and 45nm library and following parameters and error metric were estimated shown in Table 18.

Table 18: Represents the parameter and error metrics provided for the approximate multipliers

Parameters Provided	Error Metrics
Area	MAE - mean absolute error
Power	MSE - mean squared error
Delay	MRE - mean relative error
	WCE - worst case error
	WCRE - worst case relative error
	EP - error probability
	VAE - variance of absolute error

5.1.3 Datafile

Information used by GA for fitness calculation is shown in Table 19 saved in “MUL DATA” file.

Table 19: The imformation content in a “MUL Data” file

MULIPLIER INFORMATION			
BINARY NAME	MULTIPLIER NAME	AREA (180nm)	POWER (180nm)

We have assigned a binary name to all multipliers with respect to their index values these binary names are to be used in population modification process to determine the parents and their index. For our work we have used area and power parameters provided for 180nm at [38], for approximate area and power estimations in GA.

5.1.4 Working of Proposed GA tool

The workflow of the proposed GA is represented in Algorithm 7. At the first iteration, it will use the population initialization function to initialize the population and pass it to the fitness function fitness_0. While in the next iterations it will pass the modified population to fitness function fitness_1. The population is modified with respect to their individual fitness values. In order to terminate the GA two of one condition must be satisfied, the MSE value of the current population is lesser than the user-defined threshold. Second, the number generation also known as loop iteration is completed. If these conditions are not satisfied the GA will keep on working.

Algorithm 7: Proposed Genetic Algorithm

```
 Input: size , er , ar, pr , gen, thresh , α, β, γ, mul_dt, tf-approx
Output: F_ind, Mse
Procedure:
t ← 0
Po ← initialize(size, mul_dt)
While(t < gen) do:
if( t == 0) do:
   F_indt,Mset ← Fitness_0(Po , er , ar pr, α, β, γ,mul_dt, tf-approx)
Else do:
    F_indt,Mset ← Fitness_1(Po , er , ar pr, α, β, γ, indx, prt,mul_dt, tf-approx)
if(Mset <= thresh)do:
  break
indx , prt ←  Modification(F_ind, Pot) # index and parent
         Pot[indx] ← prt
         End
Return  F_ind, Mse
End procedure
```

5.1.4.1 Inputs needed by Genetic Algorithm

Following inputs are required by our proposed tool

1. Population Size required to initialize population (**size**)
2. User acceptable error in accuracy of CNN model (**er**)
3. Approximated area required by user (**ar**)
4. Approximated power required by user (**pr**)
5. Number of generations (**gen**)
6. Threshold (**thresh**)
7. Preference variables (**α, β, γ**)
8. Tf-approximate library (**tf-approx**)
9. Multipliers datafile (**mul_ft**)

5.1.4.2 Output provided by Genetic Algorithm

Our proposed tool provides following outputs

1. Fitness of final population (**MSE**)
2. Fitness of each individual in final Population (**Find**)
3. Final Population (**Po**)

5.1.4.3 Population Initialization

As mentioned above the population is initialized with respect to the type of problem the user is dealing with. In our situation, we want to determine the best set of approximate multipliers for a certain CNN model. Hence to initialize our population we need to select random multipliers from the data file. As we have given all multipliers an index number which can be represented with an integer value. So, we can generate an array of random integer values of population size (m). These generated values will represent the index of the approximate multiplier to be used in the data file. This tool is tested for two different cases. The First provides convolutional layers with the same multipliers and the second provides all convolutional layers with different multipliers in a CNN model. Population initialized for both cases is represented in Table 20.

Table 20: The structure of initial population array

Case 1: Different Multipliers for all Convolutional layers	**Case 2: Same Multipliers for all Convolutional layers**
Population = array([c1,…,cn] ,……., [cM,….,cM])	Population = array([c1], [c2],……., [cM])

Where "c" defined convolutional layers.

5.1.4.4 Fitness Evaluation

The proposed tool uses two fitness functions "Fitness_0" and "Fitness_1". At initialization, the population is processed through "Fitness_0" and then for all generations, itis processed through "Fitness_1". This is done to avoid unnecessary calculations taking a lot of time. A fitness function will call the tf-approx library to calculate approximate accuracy at runtime, inferring a CNN model. Hence, fitness_0 evaluates the model accuracy for all members in inputted population, while fitness_1 only evaluates the accuracy for modified members updating the previous individual fitness list at modified indexes only.

5.1.4.4.1 Fitness Evaluation Parameters

Fitness function is determined based on following three paraments

1. Inference Error (**Ye**)
2. Approximate Area (**Ya**)
3. Approximate Power (**Yp**)

As already discussed, before we estimate the approximate accuracy using the tf-approximate library, this is subtracted from the original accuracy of the model to determine inference error. While area and power are calculated using equations 21 and 22.

$$Area = \sum_{c=0}^{No.of\ Conv\ Layers} \frac{Parameter\ of\ layer\ c}{Total\ Paramters} \times (Area\ Mult[c]) \tag{21}$$

$$Power = \sum_{c=0}^{No.of\ Conv\ Layers} \frac{Parameter\ of\ layer\ c}{Total\ Paramters} \times (Power\ Mult\ [c]) \tag{22}$$

Where c represents convolutional layers. Total parameters represent the sum of parameters in all convolutional layers. After all, parameters are estimated, we determine the percentage absolute error for all these user-inputted required parameters with respect to reference variables.

- **Preference Variables**

  Preference variables are used tell the proposed algorithm which user defined parameter is critical. In this case we have three preference variables representing one of each evaluation parameter (**$\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$**). For example, if accuracy loss is more critical than area and power then weight given to $\boldsymbol{\alpha}$ will be greater than $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ variables. The sum of all three preference variables $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ is **1**.

Now we know all estimation parameter and formula lets discuss the internal working of Fitness function given in Algorithm 8.

Algorithm 8: Fitness_0

```
Input: Po, Er, ar, pr, datafile, α ,β , $\gamma$ ,tf-approx
Output: FP, F_ind
Procedure:
    t ← 0
    F_ind ← $\varphi$
    While(t < length (Po)) do:
        Error = tf-approx-inference (Po)
        Area = $\sum_{0}^{No.of\ Conv\ Layers\ (c)} \frac{Parameter\ of\ layer\ c}{Total\ Paramters} \times (datafile[area][c])$          (21)
        Power = $\sum_{0}^{No.of\ Conv\ Layers\ (c)} \frac{Parameter\ of\ layer\ c}{Total\ Paramters} \times (datafile[power][c])$          (22)
        Et ← abs(Error – Ee) × α #abs error
        At ← abs(Area – ar) × β #abs error
        Pt ← abs(Power – pe) × $\gamma$  #abs error
        F_ind U (Et,Pt,At)
    End
    FP = (sum(E) + sum(P) + sum(A) )$\frac{1}{n}$ #mse          (23)
    Return FP, FP_ind
End procedure
```

We already have discussed the calculation of each individual (F_ind). Final Fitness (FP) for entire population is determined using MSE at equation (23) in above pseudocode. Fitness_1 uses similar equations and code the only change is that is only calculates the fitness for modified individuals and update F_ind at that index and determine FP using same formula.

5.1.4.5 Population Modification

Population Modification is a process of selecting two or more individual to generate new offspring and updating them into current population using following three procedures:

1. Parent Selection
2. Crossover
3. Mutation

These procedures are reused from section 3.1.4.6. It takes binary label are input which represents the index of multiplier in datafile. After modifying these binary labels, we convert them to decimal and pick multipliers at that indexes in datafiles and replace them with selected parents.

5.2 Experimentations and Results

In this section, the performance of proposed module for H/w approximation (approximate multipliers) is being evaluated. The proposed genetic algorithm is implemented using Python 3.8 and cuda (11.1) for NIVIDA GPU on LINUX operating system. We performed experiments with different user requirements of accuracy loss, area, power along with preference variable (α, β and γ) and threshold values. The algorithm is tested for models and datasets defined in section 3.2.1.

5.2.1 Performance Evaluation

To determine the performance of our proposed optimizer we estimated the best approximate multipliers using following three cases shown in Table 21 for same multipliers at each convolutional layer. While the cases shown in Table 23 were used to evaluated when different multipliers are used for all convolutional layers in a model. For this we integrated tf-approximate library discussed in section 5.1.1 with our proposed algorithm to make accurate calculation for accuracy loss. While the relative area and power is calculated using equation (21) and (22) in section 5.1.4.4

5.2.1.1 Evaluation Results for same of multiplier at each convolutional layer

For testing the proposed framework, we assumed following three user cases for same multiplier at each convolutional layer shown in Table 21.

Table 21: User cases for the performance evaluation of proposed module

Parameters	Case 1	Case 2	Case 3
Iterations	50	100	200
Size	10	10	10
User Error	0.01	0.1	0.09
User Area	350.4	350.6	400.5
User Power	230	230	180
Alpha	0.6	0.6	0.6
Beta	0.2	0.3	0.2
Gamma	0.2	0.1	0.2
Threshold	0.08	0.1	0.05

5.2.1.1.1 Experimentation Results

The following Table 22 shows the results of outputted fitness values by proposed model for our final population of best multipliers.

Table 22: MSE and iteration for nest population of multipliers

Same Multipliers for All Convolutional Layers						
Model	Case 1		Case 2		Case 3	
	iter	fitness	iter	fitness	iter	fitness
LeNet 5 (MNIST)	21	0.07932563151	100	0.0997183078	19	0.0976280627
Cifar-Quick (CIFAR 10)	50	0.20852299828	100	0.1982739601	200	0.1763754142
VGG-16 (ILSVR 2012)	50	0.20211111391	100	0.1932175198	200	0.22248893164

We also plotted graphs showing the convergence of fitness value, MSE of area and power of the population at current iteration with respect to user requirements. Case one graphs are plotted for LeNet-5 Network classifying MNIST dataset shown in Figure 27(a) (b) and (c).

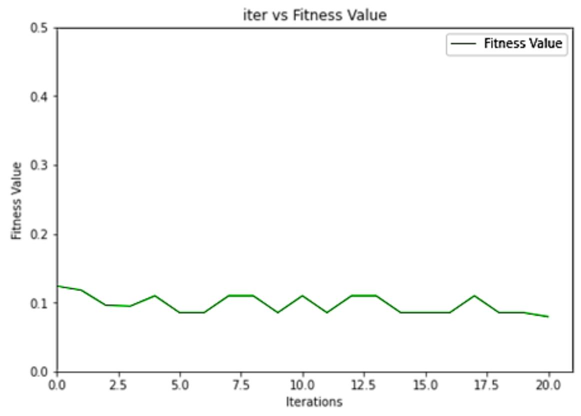


Figure 27(a): Convergence of Fitness values with iterations

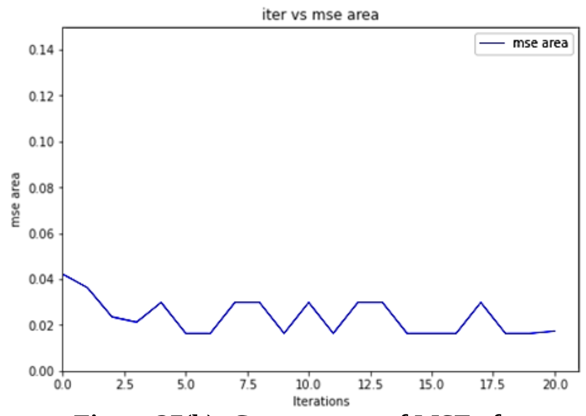


Figure 27(b): Convergence of MSE of area with iterations

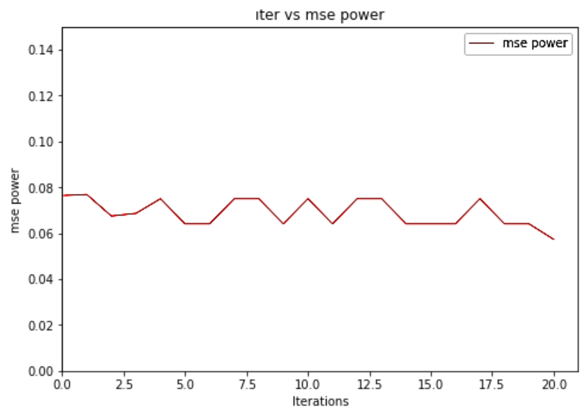


Figure 27(c): Convergence of MSE of Power with iterations

Case2 graphs are plotted for Cifar-Quick Network classifying Cifar10 dataset shown in Figure 28(a), (b) and (c).

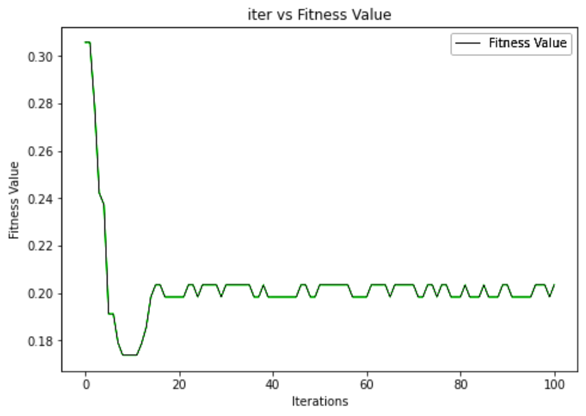


Figure 28(a): Convergence of Fitness values with iterations

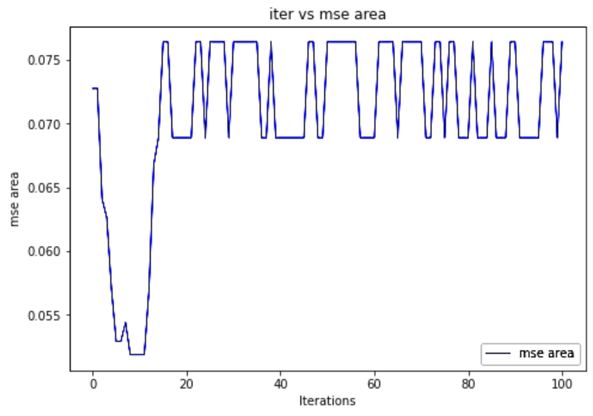


Figure 28(b): Convergence of MSE of Area with iterations

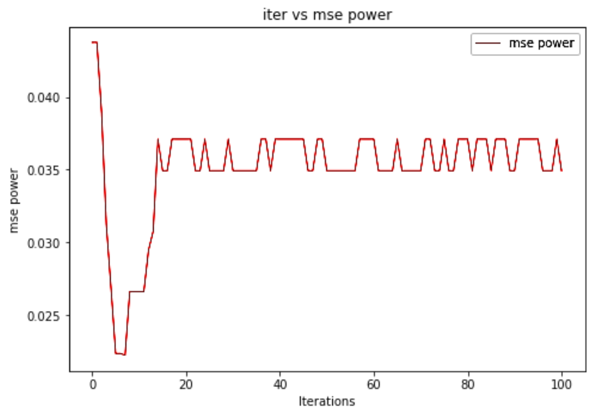


Figure 28(c): Convergence of MSE of Power with iterations

Lastly Case 3 graphs are plotted for VGG-16 Network classifying ImageNet dataset in Figure 29(a), (b) and (c)

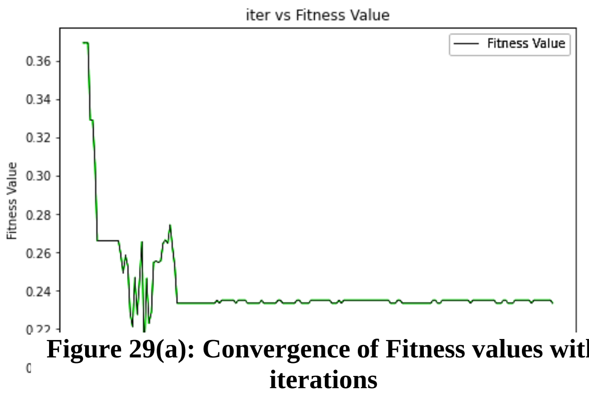


Figure 29(a): Convergence of Fitness values with iterations

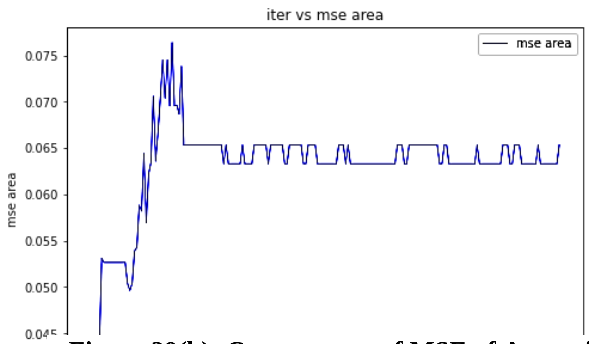


Figure 29(b): Convergence of MSE of Area with iterations

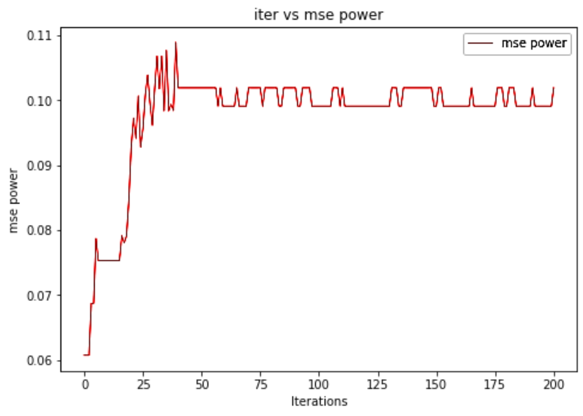


Figure 29(c): Convergence of MSE of Power with iterations

From Table 22 it is noted that CNN networks with more parameters and layers classifying complex datasets are likely to take more time to find the optimal approximate unit. As discussed in section 1, approximate multipliers introduce an error in the multiplication so for complex dataset only certain approximate units works properly with minimal loss in accuracy.

5.2.1.2 Evaluation Results for Different of multiplier at each convolutional layer

For testing the proposed framework we assumed following three user cases for different multipliers at each convolutional layer shown in Table 23.

Table 23: User cases for the performance evaluation of proposed module

Parameters	Case 1	Case 2	Case 3
Iterations	100	200	150
Size	10	10	10
User error	0.1	0.25	0.05
User area	450.4	410.4	480.5
User Power	230	210	250
Alpha	0.6	0.4	0.7
Beta	0.2	0.3	0.15
Gamma	0.2	0.3	0.15
Threshold	0.08	0.1	0.05

5.2.1.2.1 Experimentation Results

The Table 24 shows the results of outputted fitness values by proposed model for our final population of best multipliers.

Table 24: MSE and iteration for best population of multipliers

Different Multipliers for All Convolutional Layers						
Model	**Case 1**		**Case 2**		**Case 3**	
	iter	**fitness**	**iter**	**fitness**	**iter**	**fitness**
LeNet 5 (MNIST)	51	0.07932563151	174	0.0997183078	19	0.0476280627
Cifar-Quick (CIFAR 10)	100	0.20852299828	34	0.0982739601	150	0.1763754142
VGG-16 (ILSVR 2012)	100	0.20211111391	124	0.0992175198	150	0.22248893164

We also plotted graphs showing the convergence of fitness value, MSE of area and power of the population at current iteration with respect to user requirements. Case 1 graphs are plotted for LeNet-5 Network classifying MNIST dataset shown in Figure 30(a), (b) and (c).

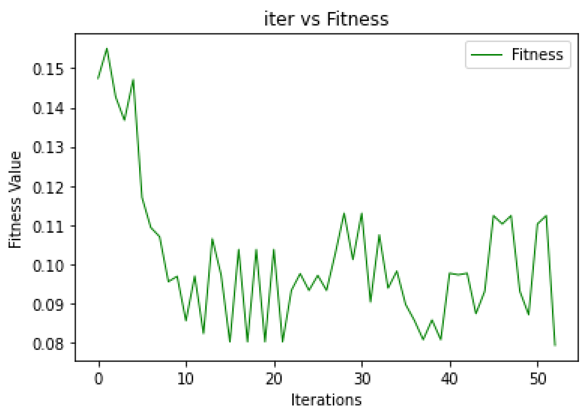


Figure 30(a): Convergence of Fitness values with iterations

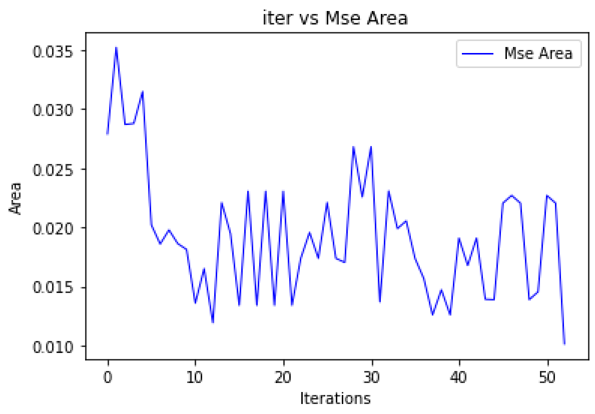


Figure 30(b): Convergence of MSE of Area with iterations

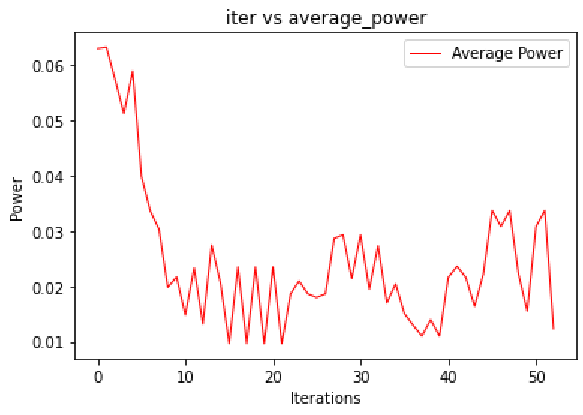


Figure 30(c): Convergence of MSE of Power with iterations

Case2 graphs are plotted for Cifar-Quick Network classifying Cifar10 dataset shown in Figure 31(a), (b) and (c).

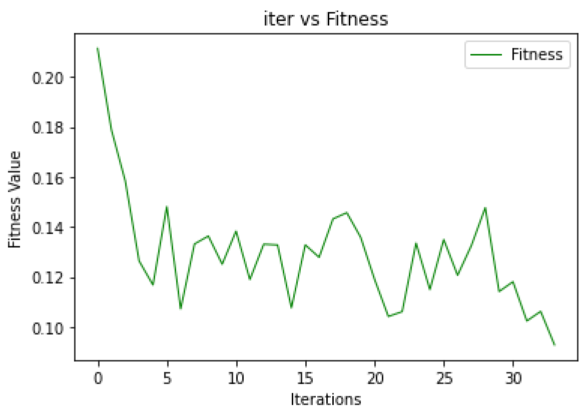


Figure 31(a): Convergence of Fitness value with iterations

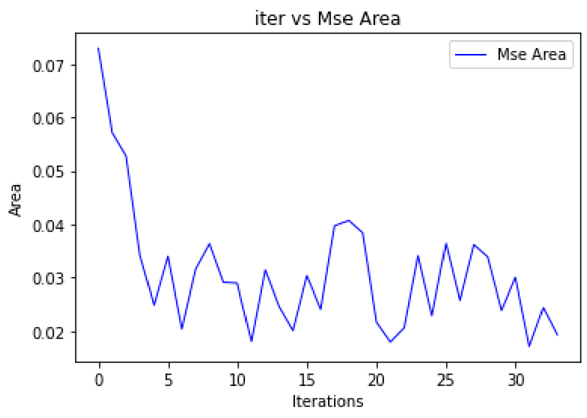


Figure 31(b): Convergence of MSE of Area with iterations

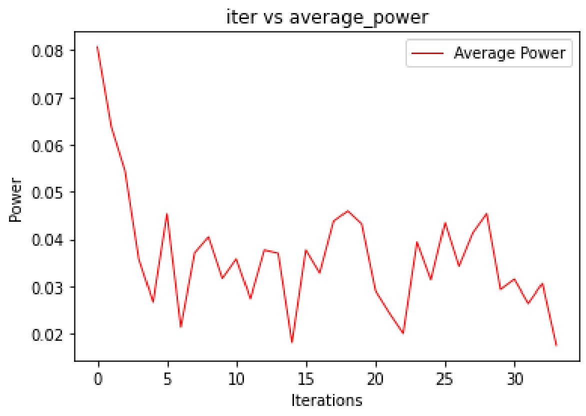


Figure 31(c): Convergence of MSE of Power with iterations

At Last, we plotted graphs for VGG-16 Network classifying ImageNet dataset for case 3 shown in Figure 32(a), (b) and (c).

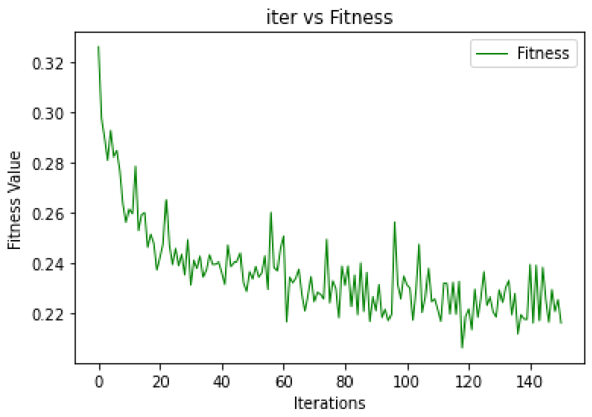


Figure 32(a): Convergence of Fitness value with iterations

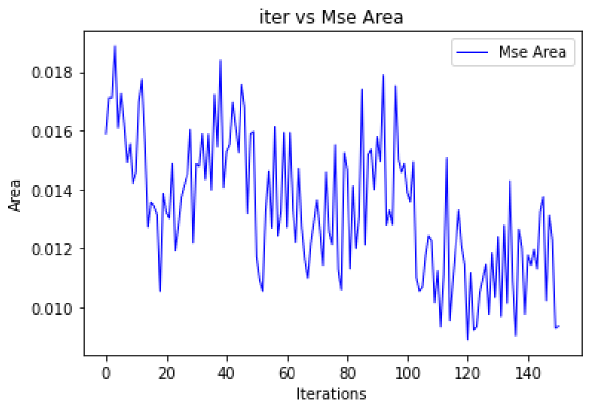


Figure 32(b): Convergence of MSE of Area with iterations

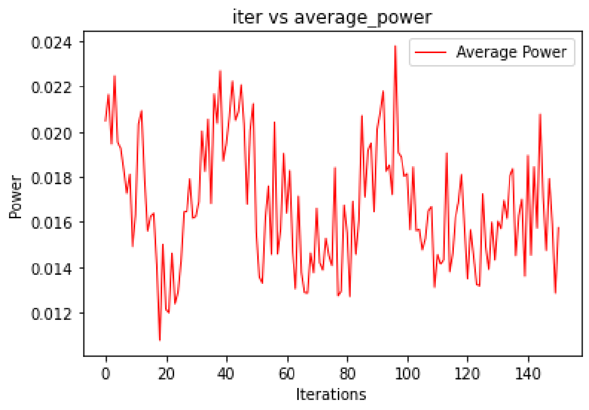


Figure 32(c): Convergence of MSE of Power with iterations

5.2.2 Final Discussion

In the previous section, we analyzed the results given by our module for the best set of multipliers. From Tables 32 and 30 we can see that CNN with more parameters and complex datasets needed a greater number of iterations to find the best result based on the user-defined cases. It is noted that LeNet-5 needs a greater number of iterations compared to VGG-16 and Cifar-Quick for case 2 from Table 31. This occurred as LeNet-5 is a simple network compared to VGG-16 and Cifar-Quick and is classifying the MNIST dataset which is also much simpler than the ImageNet dataset and Cifar-10. So, introducing extreme approximation in LeNet-5 would not affect its accuracy much compared to the other two networks.

The user error in case 2 of Table 31 is 0.25 is 25% of the error acceptance rate but LeNet5 accuracy loss is lesser than this error rate hence the MSE for LeNet-5 increases requiring more iterations to reduce the certain MSE. On other hand, VGG-16 and Cifar-Quick have greater accuracy loss. Hence, we can say that the accuracy loss due to approximations in certain CNN algorithms really depends on the complexity of the algorithm and the dataset it's working on.

5.3 Chapter Summary

This chapter deals with the optimization made at H/W levels specifically using approximate multipliers. The designed module estimates the best multipliers for the CNN model based on user requirements. The proposed module is based on GA utilizing the tf-approx library mentioned in section 5.1.1. for the inference of the CNN model using an approximate multiplier within python. The fitness is calculated based on user required area, accuracy loss, and power. The module was tested for datasets and networks mentioned in section 3.2.1. The module was able to identify the best multipliers for all CNN networks based on user-defined requirements. The said module is time efficient and is designed to avoid the inference of repeated multipliers in population to reduce execution time using two different fitness functions.

CHAPTER 6: CONCLUSION & FUTURE WORK

6.1 Conclusion

In this study, we proposed multiple tools for the optimization of CNN specifically in FPGAs. The first module deals with data compression of CNN weights to reduce memory requirements in FPGA. For our module, we used quantization and Canonical Huffman Encoding to compress the weights. The proposed tool efficiently determines the optimal parameters for Quantization i.e., N-levels to quantized weights according to user requirements without performing any inference in the CNN model. It makes use of regression equations to determine the effect of N-levels on accuracy and memory for certain CNN models. For verification of our proposed module, the estimated parameters were used to quantize CNN pre-trained weights and determine the accuracy loss and memory being saved. Then these actual values were used to determine the actual fitness of the estimated parameters. An absolute error was calculated between original MSE and estimated MSE. The highest average absolute error was 0.038 for non-uniform quantization with different N-levels for convolution and dense layers.

The second module is designed to decompress the compressed data in an FPGA. The proposed decoder uses only a single LUT to verify the codeword and manages to decode a single code within one clock cycle. The proposed design only uses 1% of system resources with a maximum frequency of 408.97MHz. The proposed decoder performs better when compared with the previously designed decoders.

The third Module deals with the optimization made at H/W levels specifically using approximate multipliers. An efficient optimizer tool for determining best multipliers units for a CNNs model. The proposed module is time efficient and returns the multipliers efficiently with respect to user requirements based on mean squared error. The tool is designed to avoid the inference of repeated multipliers in population to reduce execution time.

6.2 Future Work

The future for this research domain has following significant directions based on the modules

6.2.1 Approximate Computing

In Tool 1 a strategy can be developed to enhance the results of the regression. Regression equations can be more optimized by adding the dataset information that is to be used in the CNN model.

Further optimization of this regression equation can be done by designing some optimizer such as in module-1 to determine the best coefficients and intercept to reduce regression error.

While optimization tools can further be enhanced for determining the parameters of other used software and hardware approximate computing techniques and using different methods for population modification.

6.2.2 Hardware Decoder

In the future, we will integrate this framework with complex CNNs and other machine learning applications. A strategy can be developed to optimize LUT_valid so it does not get problematic to store in memory if it contains symbols more than 8-bits.

References

[1] Google Trends 2022. [online] Available at: <https://trends.google.com/trends/explore?date=all&q=deep%20neural%20network> [Accessed 4 April 2022].

[2] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey," *Electronics (Switzerland),* vol. 8, no. 11. 2019.

[3] M. Shahshahani, P. Goswami and D. Bhatia, "Memory Optimization Techniques for FPGA based CNN Implementations," 2018 IEEE 13th Dallas Circuits and Systems Conference (DCAS), 2018, pp. 1-6, doi: 10.1109/DCAS.2018.8620112.

[4] H. Li, H. Samet, A. Kadav, I. Durdanovic, and H. P. Graf, "Pruning filters for efficient convnets," in 5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings, 2017.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings,* 2016.

[6] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016,* 2016, pp. 243–254.

[7] A. Alqahtani, X. Xie, E. Essa, and M. W. Jones, "Neuron-based network pruning based on majority voting," in *Proceedings - International Conference on Pattern Recognition*, 2020, pp. 3090–3097.

[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* 2018, pp. 2704–2713.

[9] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApproxSb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017,* 2017, pp. 258–261.

[10] Z. Wang, M. A. Trefzer, S. J. Bale, and A. M. Tyrrell, "Approximate Multiply-Accumulate Array for Convolutional Neural Networks on FPGA," in *Proceedings - 2019 14th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip, ReCoSoC 2019*, 2019, pp. 35–42.

[11] S. Member, M. S. Kim, A. A. Del Barrio, and L. T. Oliveira, "Efficient Mitchell 's Approximate Log Multipliers for Convolutional Neural Networks," IEEE Transactions on Computers, vol. 68, no. 5, pp. 660-675, 2018

[12] J. Faraone, M. Kumm, M. Hardieck, P. Zipf, X. Liu, D. Boland, and P. H. W. Leong, "AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 1, pp. 115–128, 202 0.

[13] M. S. Kim, A. A. D. Barrio, R. Hermida, and N. Bagherzadeh, "Low-power implementation of Mitchells approximate logarithmic multiplication for convolutional neural networks," 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), 2018.

[14] T. Kowsalya, "Area and Power Efficient Pipelined Hybrid Merged Adders for Customized Deep Learning Framework for FPGA Implementation," Microprocess. Microsyst., p. 102906, 2019.

[15] L. Luo, Z. Chen, X. Yang, F. Qiao, Q. Wei, and H. Yang,"A single clock cycle approximate adder with hybrid prediction and error compensation methods," Microelectronics J., vol. 87, no. September 2018, pp.45–50, 2019.

[16] C. Guo, L. Zhang, X. Zhou, W. Qian, and C. Zhuo, "A Reconfigurable Approximate Multiplier for Quantized CNN Applications," In 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 235-240, 2020.

[17] H. O. Ahmed, M. Ghoneima, and M. Dessouky, "Concurrent MAC unit design using VHDL for deep learning networks on FPGA," in ISCAIE 2018 - 2018 IEEE Symposium on Computer Applications and Industrial Electronics, 2018, pp. 31–36.

[18] T. Guan, P. Liu, X. Zeng, M. Kim, and M. Seok, "Recursive Binary Neural Network Training model for Efficient Usage of On-Chip Memory," IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 1–13, 2019.

[19] Q. Huang, K. Zhou, S. You, and U. Neumann, ''Learning to prune filters in convolutional neural networks,'' in Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV), Mar. 2018, pp. 709–718.

[20] A. Aghasi, A. Abdi, and J. Romberg, ''Fast convex pruning of deep neural networks,'' SIAM J. Math. Data Sci., vol. 2, no. 1, pp. 158–188, Jan. 2020.

[21] S. A. Dahri, A. F. Chandio, and N. A. Zardari, ''Implementation of Huffman decoder on fpga,'' Int. J. Eng. Res. Appl., vol. 6, no. 1, pp. 84–88, Jan. 2016. [Online]. Available: www.ijera.com

[22] M. F. Mansour, ''Efficient Huffman decoding with table lookup,'' in Proc. IEEE Int. Conf. Acoust., Speech Signal Process. - ICASSP, Apr. 2007, pp. II–53–II–56.

[23] M.-B. Lin and Y.-Y. Chang, ''A new architecture of a two-stage lossless data compression and decompression algorithm,'' IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 17, no. 9, pp. 1297–1303, Sep. 2009.

[24] T. Malach, S. Greenberg and M. Haiut, "Hardware-Based Real-Time Deep Neural Network Lossless Weights Compression," in IEEE Access, vol. 8, pp. 205051-205060, 2020, doi: 10.1109/ACCESS.2020.3037254.

[25] Z. Aspar, Z. M. Yusof, and I. Suleiman, “Parallel Huffman decoder with an optimize Look Up Table option on FPGA,” in IEEE Region 10 Annual International Conference, Proceedings/TENCON, 2000, vol. 1, doi: 10.1109/tencon.2000.893543.

[26] J. Carlos Angulo, A. Carlos Fajardo, O. M. Reyes, and V. Javier Castillo, “FPGA implementation of a huffman decoder for high speed seismic data decompression,” in Data Compression Conference Proceedings, 2014, p. 396, doi: 10.1109/DCC.2014.83.

[27] S. M. Najmabadi, H. S. Tungal, T.-H. Tran, and S. Simon, ‘‘Hardwarebased architecture for asymmetric numeral systems entropy decoder,’’ in Proc. Conf. Design Archit. Signal Image Process. (DASIP), Sep. 2017, pp. 1–6.

[28] W. Nogami, T. Ikegami, S. I. O’Uchi, R. Takano, and T. Kudoh, “Optimizing Weight Value Quantization for CNN Inference,” in *Proceedings of the International Joint Conference on Neural Networks*, 2019, vol. 2019–July.

[29] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, “Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 2, pp. 317–328, 2020.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278–2323, 1998.

[31] L. Deng, “The MNIST database of handwritten digit images for machine learning research,” IEEE Signal Process. Mag., vol. 29, no. 6, pp. 141–142, 2012, doi: 10.1109/MSP.2012.2211477.

[32] Y. Zhou, S. Song, and N. M. Cheung, “On classification of distorted images with deep convolutional neural networks,” in ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, 2017, pp. 1213–1217.

[33] A. Krizhevsky and G. Hinton, “Convolutional deep belief networks on cifar-10,” *Unpubl. Manuscr.*, pp. 1–9, 2010.

[34] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[35] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A simple framework for contrastive learning of visual representations,” in *37th International Conference on Machine Learning, ICML 2020*, 2020, vol. PartF168147-3, pp. 1575–1585.

[36] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, “TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU,” in *Proceedings of the 2020 Design,*

Automation and Test in Europe Conference and Exhibition, DATE 2020, 2020, pp. 294–297. doi: 10.23919/DATE48585.2020.9116299.

[37] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, “ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining,” in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2019, vol. 2019–November. doi: 10.1109/ICCAD45719.2019.8942068.

[38] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, “EvoApproxSb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods,” in *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, 2017, pp. 258–261. doi: 10.23919/DATE.2017.7926993.