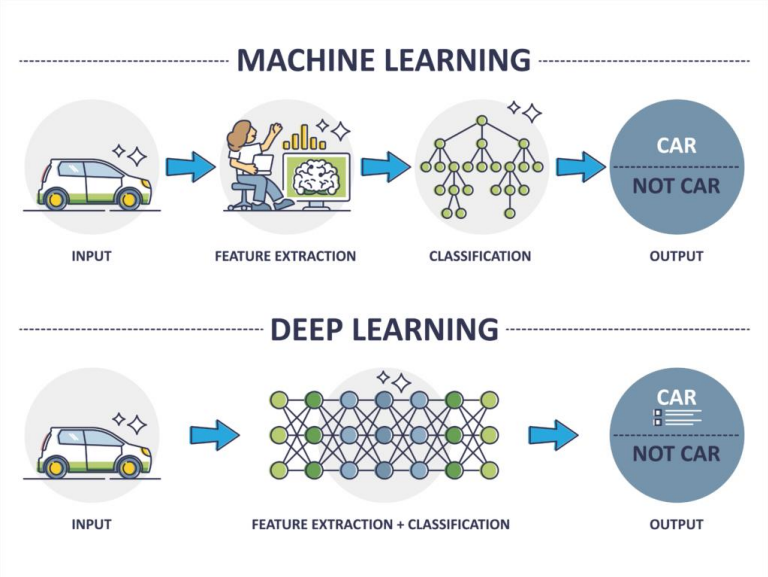
Masters Project Report on Compression of Neural Networks

Sreyan Biswas
19EC39036
Indian Institute of Technology Kharagpur

TABLE OF CONTENTS

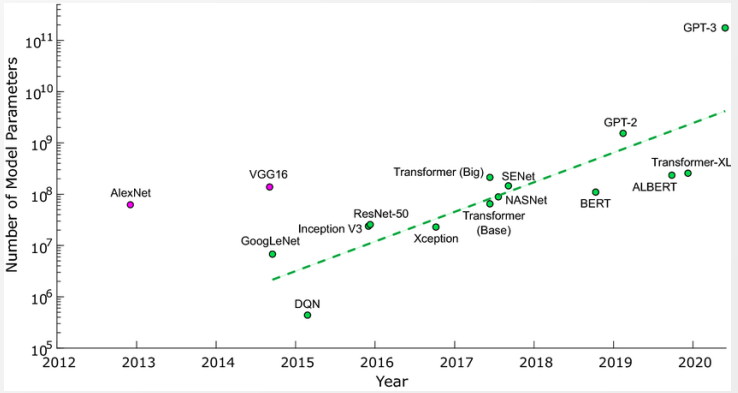
1.1 Introduction

- Machine Learning and its subset, Deep Learning, are highly prominent, especially in Computer Vision and Image Recognition
- Rise of Deep CNNs Since 2012:
  - Emergence of AlexNet in 2012 marked a turning point. Deep CNNs showcase high accuracy in diverse cognitive tasks, sparking substantial research interest
- Growing demand for enhanced CNN performance prompts exploration of custom hardware accelerators.
- Field Programmable Gate Arrays (FPGAs) emerge as a compelling solution.
- Compression of neural networks becomes paramount for optimizing inference on hardware accelerators.
- Efficient resource utilization and faster computation are key benefits.



1.2 Motivation

- Storage Capacity Challenge:
  - DNN models demand extensive storage due to numerous parameters.
  - Compression efficiently utilizes storage, enabling deployment on Resource Constraint Devices (RCDs).
- Computation Requirements Hurdle:
  - Abundance of Floating Point Operations (FLOPs) in DNN operations can overwhelm limited computational capacity of RCDs.
  - Compression reduces computational requirements, aligning complexity with resource-constrained device capabilities.
- Latency as a Concern:
  - High training and inference times impact real-time performance.
  - Compression mechanisms enhance earliness in both training and inference phases, improving real-time application responsiveness.
- Energy Consumption Challenge:
  - DNN compression contributes to reduced energy consumption in data processing.
  - Compressed models are well-suited for deployment on battery-operated IoT devices, enhancing energy efficiency

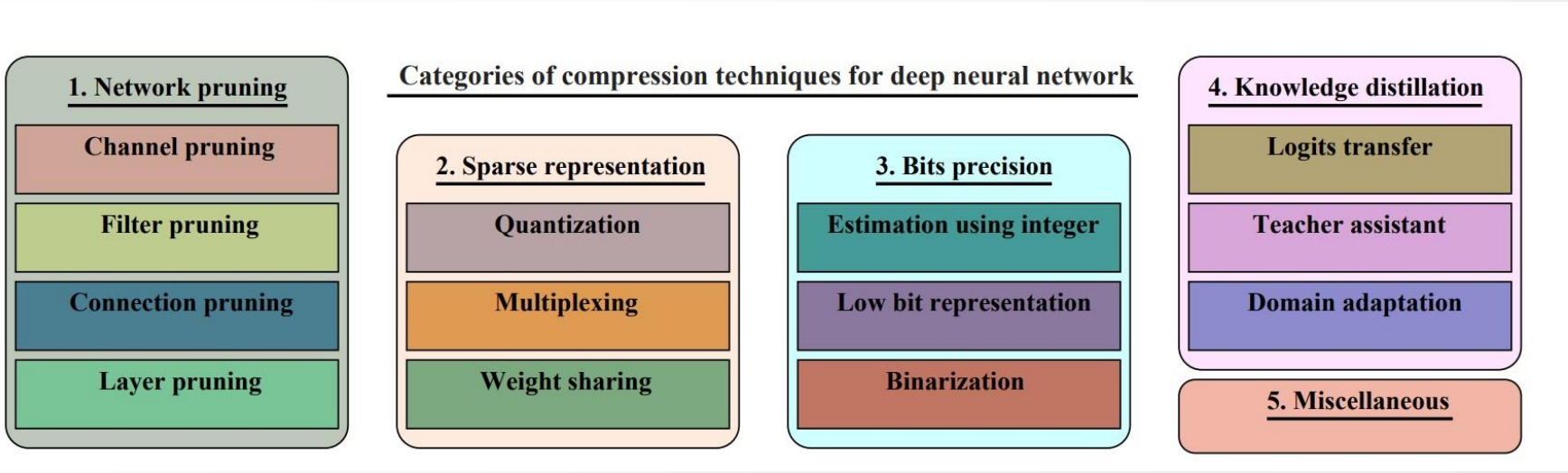


2 Background Research

The existing research on Deep Neural Network (DNN) compression can be categorized into five key categories, each derived from an extensive literature review on compression methods:

- Network Pruning:
  - Involves the removal of redundant or less important connections from the DNN.
  - Aims to reduce complexity and computational burden by trimming unnecessary network connections.
- Sparse Representation:
  - Focuses on representing DNN weights and activations in a sparse manner.
  - Aims to minimize storage requirements by leveraging sparsity, i.e. the fact that most weights can be zero.
- Bits Precision:
  - Aims to reduce the number of bits used to represent DNN weights and activations.
  - Results in smaller model sizes and faster computations by employing lower precision.
- Knowledge Distillation:
  - Involves transferring knowledge from a larger, more accurate DNN (teacher) to a smaller, less accurate DNN (student).
  - Enhances the student's performance by distilling valuable knowledge from the teacher network.

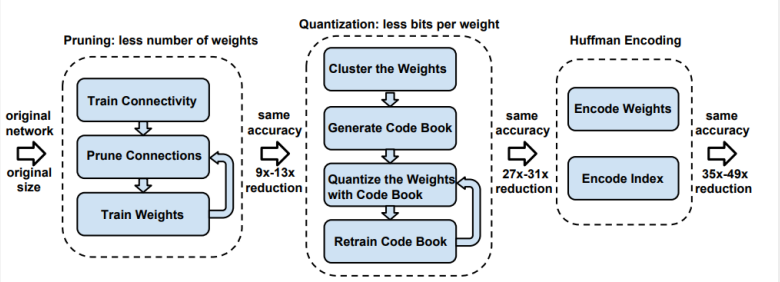
2 Categories of Compression Techniques



3 Methodology

We implement a three-stage pipeline for compressing deep neural networks:

- First, we prune the networking by removing the redundant connections, keeping only the most informative connections.
- Next, the weights are quantized so that multiple connections share the same weight, thus only the codebook (effective weights) and the indices need to be stored.
- Finally, we apply Huffman coding to take advantage of the biased distribution of effective weigh



3 Network Pruning

- Initial Training: Commence by establishing connectivity through standard network training.
- Train the neural network to establish initial connections and learn representations.
- Subsequently, we undertake pruning to remove connections with weights below a defined threshold.
- Certain weights are set to zero based on a defined criterion (e.g., low magnitude). This results in a sparse neural network.
- Pruning Techniques:
    - prune_by_percentile: Prunes weights based on a given percentile value. Identify and remove connections with weights falling below the specified percentile.
    - prune_by_std: Prunes weights based on a sensitivity value multiplied by the standard deviation of a layer's weights. Utilize the standard deviation of weights to determine sensitivity for pruning.

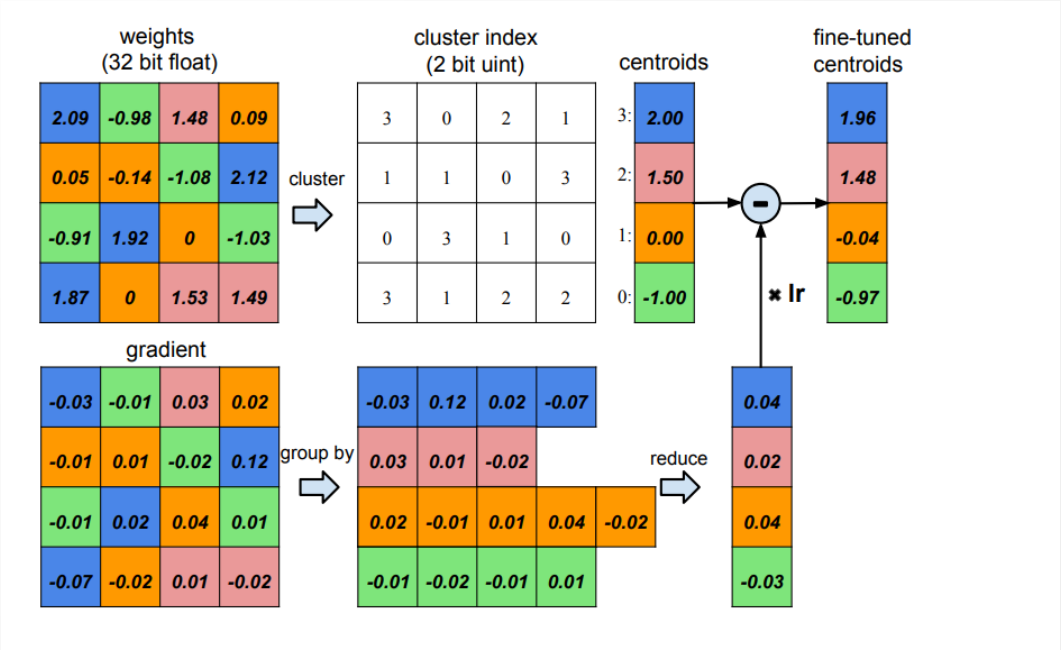
3 Quantization

- Further compress the pruned network by reducing the number of bits needed to represent each weight.
- Allow many connections to share the same weight to reduce the number of effective weights.
- Fine-tune shared weights to maintain accuracy.
- Trained quantization process:
  - Prune the network to remove unimportant connections.
  - Quantize weights to a smaller number of bits in the pruned network.
  - Train the quantized network on training data.
  - Evaluate the trained quantized network on test data.
- Compression rate calculation: Given k clusters, log2(k) bits encode the index. If n connections use b bits each, limiting connections to have a maximum of k shared weights yields the compression rate:

$$\text{Compression Rate} = \frac{k \cdot b + \log_2(k)}{n \cdot b}$$

3 K-Means Clustering

Illustration of weight quantization using clustering



3 K-Means Clustering

- We reduce storage requirements using k-means clustering to group similar weights within each layer of the trained network.
- One-dimensional k-means clustering results in centroids that serve as shared weights within the layer.
- WCSS (Within-Cluster Sum of Squares):
  - It is the sum of the squared distances between all weights in a cluster and its centroid.
  - Minimizing WCSS ensures that weights within each cluster are as similar as possible.

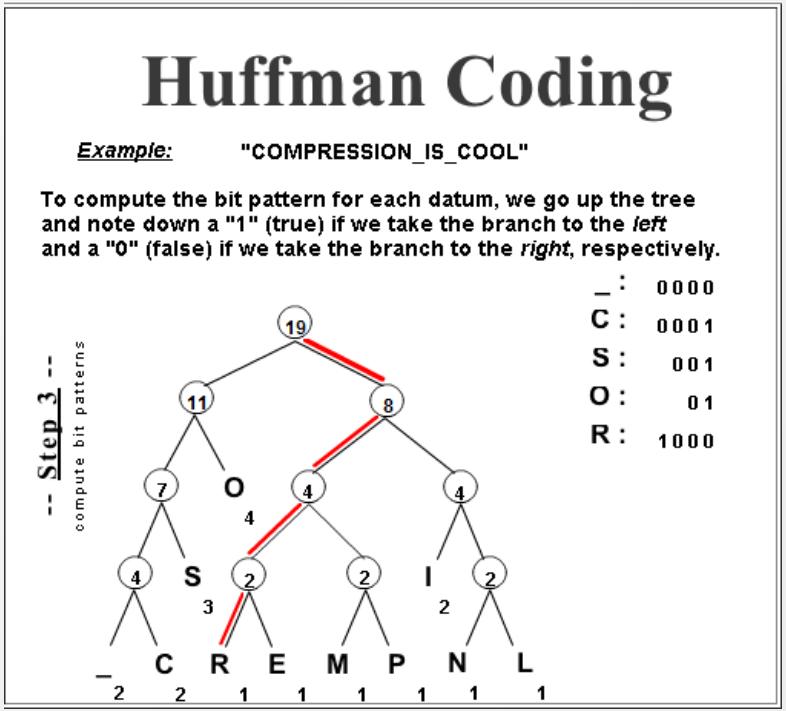
$$\arg\min_{C} \sum_{k=1}^{K} \sum_{w \in c_k} ||w - c_k||^2$$

$$\frac{\partial L}{\partial C_k} = \sum_{i,j} \frac{\partial L}{\partial W_{ij}} 1(I_{ij} = k)$$

- The first equation is the objective function that is minimized by the k-means clustering algorithm. The goal of the algorithm is to partition the n original weights W into k clusters so that the within-cluster sum of squares (WCSS) is minimized.
- The second equation is the gradient of the centroids.

3 Huffman Coding

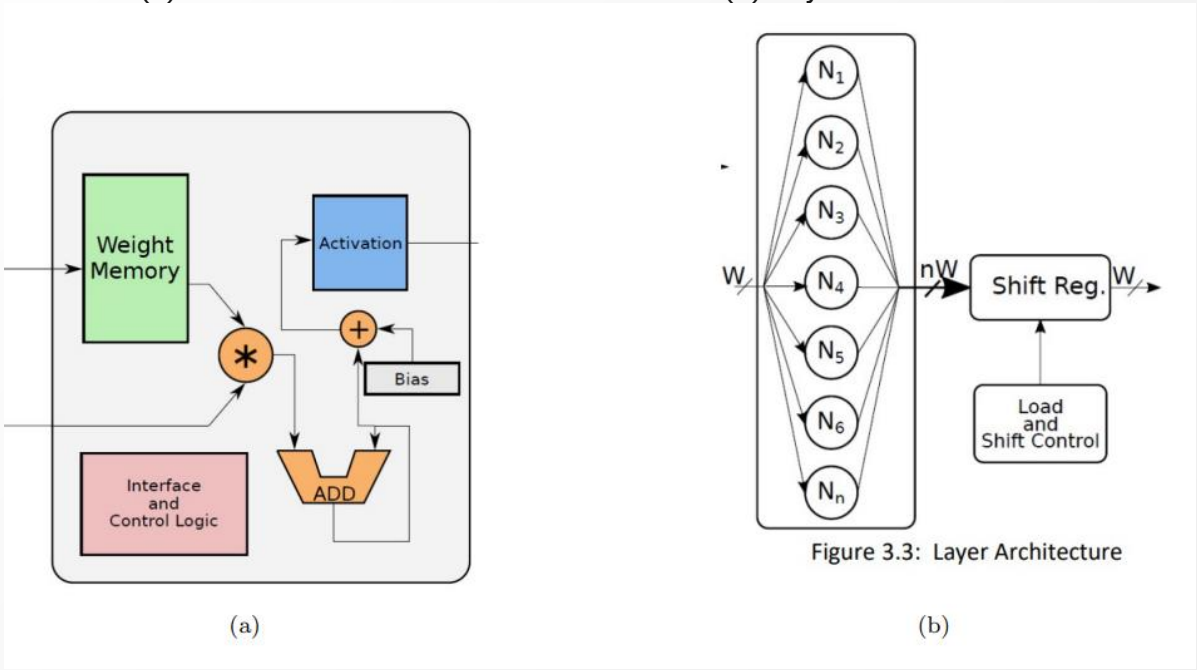
- Huffman coding is a lossless data compression algorithm used to further compress the network.
- It assigns shorter codewords to more frequent symbols and longer codewords to less frequent symbols.
- This technique is particularly effective for compressing quantized weights and sparse matrix indices, which often exhibit biased distributions.



3 Neuron and Layer Architecture

(a) Neuron Architecture

(b) Layer Architecture



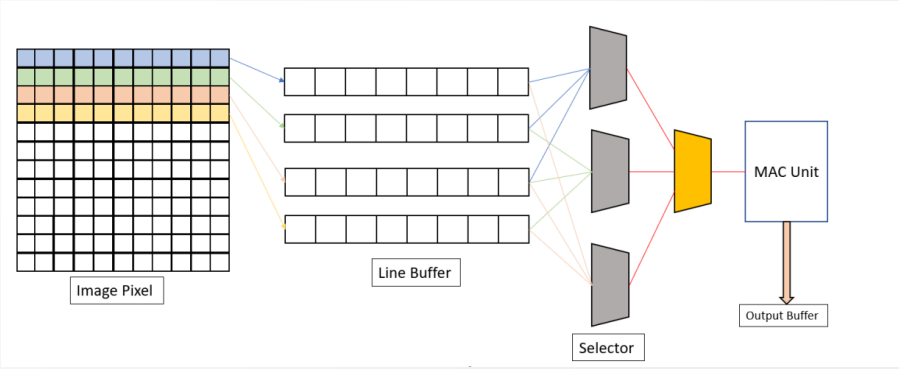
3 Neuron Architecture

- Each neuron, regardless of predecessor count, features a single data input interface.
- Internal memory, which is sized by the neuron’s input count, stores weight values. Depending on the network configuration (pre-trained or not), a RAM or a ROM with initialized weights is instantiated.
- As inputs flow in, control logic reads corresponding weights from memory, and a MAC unit multiplies and accumulates these values, adding the bias.
- Bias values, like weights, are stored in registers for pre-trained networks or configured at runtime.
- As inputs flow in, control logic reads corresponding weights from memory.
- Weight values are then used in multiplication and accumulation (MAC) operations.
- MAC Unit: Performs Multiply-Accumulate (MAC) operations.
- Multiplies input values by corresponding weights and accumulates the results. Adds bias values stored in registers for pre-trained networks or configured at runtime.
- Activation Unit: Processes the output from the MAC unit. Implemented in a look-up-table-based (LUT) or circuit-based function based on the configured activation function (e.g., Sigmoid, ReLU).

3 Convolution in Hardware

- To perform the convolution process, the structure has four modules:
    1. Line Buffer
    2. Multiply and Accumulate Unit
    3. Control Unit
    4. Top Module

- The Line Buffer functions as a temporary memory storage unit during operations, specifically holding an entire row of image data before transmitting it to the Multiply Accumulate Unit (MAC)
- The control module is responsible in maintaining the order in which the line buffers are filled. The first step is that all the three-line buffers get filled before we can start operation using the MAC unit.
- Simultaneously the fourth line buffer gets filled, to exploit the philosophy parallelism. After that as we complete the convolution operation for a single row, the control module sends the next group of three-line buffers while the first one gets filled.



4.1 Results

Optimizations defined earlier were implemented on a LeNet network and by training and evaluating on the MNIST data set using PyTorch. After the initial training the accuracy was recorded to be 97%

```
Test set: Average loss: 0.0823, Accuracy: 9762/10000 (97.62%)
--- Before pruning ---
fc1.weight | nonzeros = 235200 /235200 (100.00%) | total_pruned = 0 | shape = (300, 784)
fc1.bias | nonzeros =     300 / 300 (100.00%) | total_pruned = 0 | shape = (300,)
fc2.weight | nonzeros =   30000 / 30000 (100.00%) | total_pruned = 0 | shape = (100, 300)
fc2.bias | nonzeros =     100 / 100 (100.00%) | total_pruned = 0 | shape = (100,)
fc3.weight | nonzeros =    1000 / 1000 (100.00%) | total_pruned = 0 | shape = (10, 100)
fc3.bias | nonzeros =      10 / 10 (100.00%) | total_pruned = 0 | shape = (10,)
alive: 266610, pruned : 0, total: 266610, Compression rate :1.00x  (  0.00% pruned)
```

LISTING 5.1: Before Pruning

4.1 Results

After applying pruning, the accuracy drops to 37.4%

```
Pruning with threshold : 0.10118481516838074 for layer fc1
Pruning with threshold : 0.14223457872867584 for layer fc2
Pruning with threshold : 0.24343585968017578 for layer fc3
Test set: Average loss: 1.6647, Accuracy: 3470/10000 (34.70%)
--- After pruning ---
fc1.weight | nonzeros = 14770 / 235200 (6.28%) | total_pruned =  220430 | shape = (300, 784)
fc1.bias | nonzeros = 300 / 300 (100.00%)|total_pruned = 0|shape = (300,)
fc2.weight | nonzeros =  1953 /30000 (  6.51%) | total_pruned =   28047 | shape = (100, 300)
fc2.bias | nonzeros =  100 /100 (100.00%) | total_pruned =  0 | shape = (100,)
fc3.weight | nonzeros = 47 / 1000 (  4.70%) | total_pruned =  953 | shape = (10, 100)
fc3.bias | nonzeros =   10 / 10 (100.00%) | total_pruned =   0 | shape = (10,)

alive: 17180, pruned : 249430, total: 266610, Compression rate :   15.52x  ( 93.56% pruned)
```

4.1 Results

After retraining using the quantized weights the accuracy is restored to 97.8%

```
Test set: Average loss: 0.0751, Accuracy: 9782/10000 (97.82%)
```

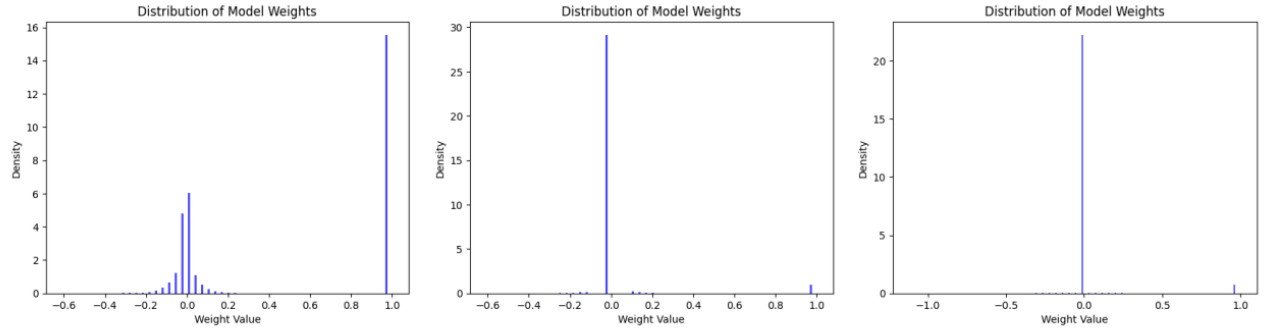
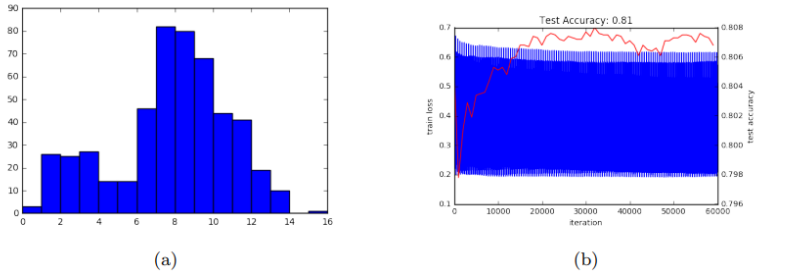


FIGURE 5.1: Distribution of weights: (a) Before Pruning (b) After Pruning (c) Retraining after Pruning

Initially the weights are varied. Then the variance reduces and most common weight is selected. Then after training the important weights are automatically learned

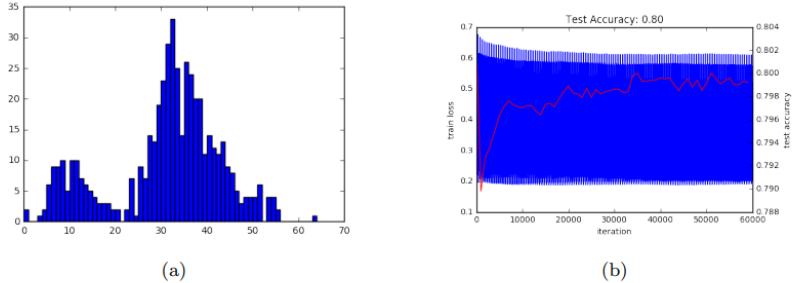
4.1 Results

Now implementing the network on a CIFAR10 dataset. The results after varying the number of clusters is



(a) (b)

FIGURE 5.2: Model trained on CIFAR10 dataset. Number of Clusters set to 16: (a) Weight distribution (b) Accuracy and Loss Plot



(a) (b)

FIGURE 5.3: Model trained on CIFAR10 dataset. Number of Clusters set to 64: (a)Weight distribution (b) Accuracy and Loss Plot

More clusters doesn't necessarily mean higher accuracy. That is the crux of the reason why quantization works well to maintain the accuracy even with limited number of weights.

4.1 Results

The time for convolution reduced from 5ms to 2.5ms when increasing the number of line buffers. Therefore Increasing the number of line buffers or expanding the number of hardware components can enhance parallelism and subsequently reduce computation time.

FIGURE 5.7: Waveform of the Testbench when three-line buffers are used

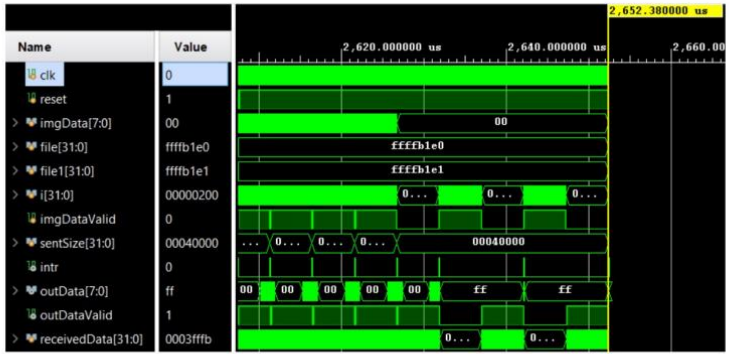


FIGURE 5.8: Waveform of the Testbench when four-line buffers are used

4.5 Future work

3D Convolution Extension:

- Explore extending 2D convolution operations to 3D for implementing neural networks on volumetric data.
- Offers opportunities to expand application possibilities beyond traditional 2D convolution.

Complex CNN Model Implementation:

- The current hardware acceleration and optimization techniques provide a foundation for implementing complete CNN models such as AlexNet, VGGNet, ResNet, etc., on FPGA.
- Resource constraints may necessitate further optimization or alternative hardware solutions for larger models.

C to Verilog Code Translation:

- Utilize C programming for neural network implementation instead of python.
- Translate C code into Verilog using Xilinx Vivado tools, facilitating a seamless transition between high-level programming and hardware description.

4.5 Future work

- There is ongoing work on implementing neural networks in C, with a focus on implementing LeNet in Verilog.

More Hardware Optimization Techniques:

- There are plans to develop optimization techniques specific to hardware implementation.
- Aim to unlock the full potential of FPGA resources during neural network execution, enhancing efficiency and overall performance.
- Reflects a progressive approach to streamline development processes and bridge the gap between software and hardware.
- Implement FINN: A Framework for Fast, Scalable Binarized Neural Network Inference in FPGA and carry out potential optimization

THANK YOU!