Compression of Deep Neural Networks

Project report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilmentt for the award of the degree of

Masters of Technology

in

Electronics and Electrical Communication Engineering
(Specialization in Vision and Intelligent Systems)

by

Sreyan Biswas

(19EC39036)

Under the supervision of

Professor Indrajit Chakrabarti

Department of Electronics and Electrical Communication Engineering

Indian Institute of Technology Kharagpur

Autumn Semester, 2023-24

November 29, 2023

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: November 29, 2023 (Sreyan Biswas)
Place: Kharagpur (19EC39036)

DEPARTMENT OF ELECTRONICS AND ELECTRICAL COMMUNICATION ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

KHARAGPUR - 721302, INDIA

CERTIFICATE

This is to certify that the project report entitled "**Compression of Deep Neural Networks** " submitted by **Sreyan Biswas** (Roll No. 19EC39036) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Masters of Technology in Electronics and Electrical Communication Engineering (Specialization in Vision and Intelligent Systems) is a record of bona fide work carried out by him under my supervision and guidance during Autumn Semester, 2023-24.

Date: November 29, 2023

Place: Kharagpur

Professor Indrajit Chakrabarti
Department of Electronics and Electrical
Communication Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India

Abstract

Name of the student: **Sreyan Biswas** Roll No: **19EC39036**
Degree for which submitted: **Masters of Technology**
Department: **Department of Electronics and Electrical Communication Engineering**
Thesis title: **Compression of Deep Neural Networks**
Thesis supervisor: **Professor Indrajit Chakrabarti**
Month and year of thesis submission: **November 29, 2023**

Machine Learning and its sub-discipline, Deep Learning, have witnessed significant popularity, particularly in Computer Vision and Image Recognition tasks. Convolutional Neural Networks (CNNs) employed in Deep Learning train weights and biases across network layers to recognize crucial image features. Since the emergence of AlexNet in 2012, Deep CNNs have garnered substantial interest, exhibiting high accuracy in various cognitive tasks and becoming a focal point for research. As the demand for CNN performance continues to rise, custom hardware accelerators, particularly Field Programmable Gate Arrays (FPGAs), present a compelling solution.

In this context, the significance of compressing neural networks becomes paramount. Compressing neural networks is crucial for optimizing inference on hardware accelerators like FPGAs, as it enables efficient resource utilization and faster computation. Therefore, this work explores a Hardware-Software co-design approach to accelerate CNN inference on an FPGA. The objective is to develop a streamlined deployment flow for CNNs on FPGAs, emphasizing the importance of compression for achieving rapid and efficient inference.

Acknowledgements

I place on record, my sincere thanks to Prof. Indrajit Chakrabarti, supervisor for this project, for his efficacious advice, perpetual and prolific encouragement, and constructive criticism during the course of the work and in the preparation of this project report. Additionally, I extend my heartfelt appreciation to Sayantan Dutta, a Ph.D. fellow under Prof. Chakrabarti, whose guidance and support were invaluable throughout my project. I would also like to express my sincere gratitude to all the staff of this department for their help rendered in the completion of this project-work. Last but not least, I extend my gratitude to all the professors in my department who have been teachers in the courses that I attended during the pursuit of my academic degree.

Contents

Chapter 1

Introduction

The proliferation of machine learning-based Internet of Things (IoT) applications has been remarkable in recent years, facilitated by the accessibility of small computing devices [21], [35]. As machine learning methods reliant on handcrafted features grapple with challenges when encountering novel datasets, Deep Neural Network (DNN) models have emerged as dominant forces in various application domains, including smart homes, agriculture, and mobility recognition . Offering automatic feature extraction from large raw data, DNN models, comprising Convolutional Neural Networks (CNNs), Fully Connected (FC) layers, and Recurrent Neural Networks (RNNs), deliver superior performance in classification, regression, and prediction tasks . Resource Challenges and the Case for Compression Despite their prowess, DNN models demand substantial resources, including energy, processing capacity, and storage, limiting their suitability for Resource Constraint Devices (RCDs). This resource-intensive nature poses challenges for real-time inference and deployment in browser-based applications.

1.1 Motivation

In the dynamic landscape of neural networks, a myriad of challenges has emerged, acting as catalysts propelling the quest for innovative solutions in network compression. These challenges, each presenting a unique hurdle, underscore the pressing need to streamline and enhance the efficiency of neural networks. A glimpse into some of these formidable challenges provides insight into the motivation behind the relentless pursuit of compression methodologies:

- **Storage Capacity:** DNN models, with their vast number of parameters, demand considerable storage, hindering deployment on Resource Constraint Devices (RCDs). Compression allows efficient storage utilization, preserving resources for deployment on these devices.
- **Computation Requirements:** The abundance of Floating Point Operations (FLOPs) in DNN operations can surpass the limited computational capacity of RCDs [30]. DNN compression, by reducing computational requirements, aligns model complexity with the capabilities of resource-constrained devices.
- **Latency:** The training and inference times of DNN models can be significantly high, impacting real-time performance. Compression mechanisms offer a higher degree of earliness in both training and inference phases, enhancing real-time application responsiveness.
- **Privacy:** Transmitting data from the source to high-end machines can compromise security and privacy. In-situ processing using compressed DNN models on RCDs not only ensures privacy but also enhances data security.
- **Energy Consumption:** DNN compression contributes to reduced energy consumption in data processing . This feature makes compressed models well-suited for deployment on battery-operated IoT devices, enhancing their energy efficiency.

Chapter 2

Objective and Problem Definition

This work aims to explore and implement a Hardware-Software co-design approach for accelerating Convolutional Neural Network (CNN) inference. The overarching goal is to develop a streamlined deployment flow that prioritizes speed and efficiency, addressing the escalating performance demands of CNNs. Specific objectives include:

- Exploring compression techniques for neural networks to optimize inference on FPGA-based hardware accelerators, thereby improving resource utilization.

- Integrate insights from an extensive literature review on neural network compression to inform and enhance the design process.

- Investigating the potential of custom hardware accelerators, with a focus on FPGAs, as a viable solution for enhancing CNN performance.

- Evaluating the proposed approach's effectiveness in achieving rapid inference while maintaining high accuracy in tasks related to cognition.

- Assessing the overall impact of compression techniques on the performance and resource utilization of CNNs on FPGA platforms.

Chapter 3

Background Research

3.1 Introduction

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:

1. **Network Pruning:** Involves the removal of redundant or less important connections from the DNN to reduce complexity and computational burden.
2. **Sparse Representation:** Focuses on representing DNN weights and activations in a sparse manner to minimize storage requirements.
3. **Bits Precision:** Aims to reduce the number of bits used to represent DNN weights and activations, leading to smaller model sizes and faster computations.
4. **Knowledge Distillation:** Involves transferring knowledge from a larger, more accurate DNN (teacher) to a smaller, less accurate DNN (student) to improve the student's performance.

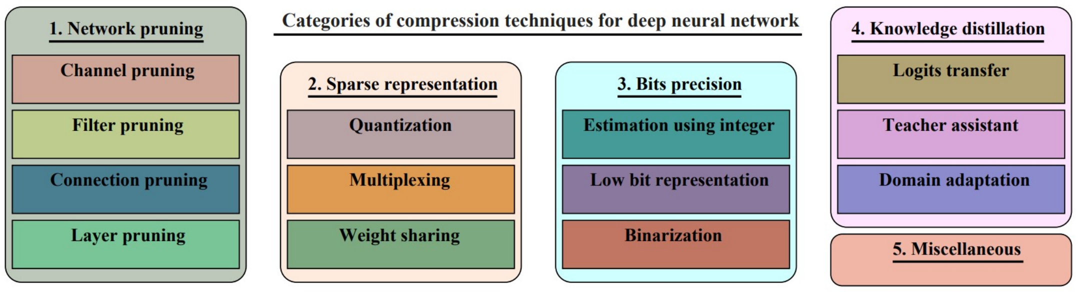


FIGURE 3.1: Categories of Compression Techniques

3.2 Pruning

3.2.1 Channel Pruning

Channel pruning, a technique for compressing convolutional neural networks (CNNs), involves the removal of entire channels from the network. Various techniques for channel pruning are discussed in the paper, each with its distinct methodology:

1. Regression-based Channel Selection[17]: Utilizes regression to predict reconstruction errors for each channel. Channels with the highest reconstruction error are removed.
2. Improved Pruning Operation[25]: Enhances regression-based channel selection by determining a pruning threshold. The following equation represents the minimization problem that is solved by the channel pruning method:

$$\arg\min_{\delta,F} \frac{1}{N}\|O - \sum_{j=1}^{k} \delta_j F_j I_j\|_F^2,$$
$$\text{subject to } \|\delta\|_0 \leq k',$$

where: O is the output matrix of the convolutional layer, F_j is the j-th slice of the convolutional filter, I_j is the j-th channel of the input data, δ is a coefficient vector of length k used in channel selection, $\|\cdot\|_0$ is the L0-norm, which counts the number of non-zero elements in a vector, $\|\cdot\|_F$ is the Frobenius norm. The goal of the problem is to find a sparse coefficient vector δ that selects a subset of channels to keep, while minimizing the reconstruction error between the output of the convolutional layer and the original input data. The constraint $\|\delta\|_0 \leq k'$ ensures that the number of selected channels does not exceed k'.

3. Evolutionary NetArchitecture Search (EvoNAS) [3]: Challenges the assumption of independence among channels and neurons in deep neural networks. Formulates a combinatorial optimization problem for network pruning, solved using an evolutionary algorithm and attention mechanism.
4. Network Slimming[27]: Introduces a scaling factor (Δ) for each channel. Jointly trains the model and Δ using sparsity regularization to enforce channel-level sparsity.
5. MobileNets [18]: Estimates hyperparameters (width multiplier and resolution multiplier) for resource adaptation in different applications. Applies depthwise convolution with a single filter to each input channel. Achieves faster inference by exploiting dataset sparsity. Width multiplier (wd) thins the network uniformly at each layer, reducing the number of input channels (M) to (wd)M. Resolution multiplier (rm) is applied to the input image, subsequently reducing the internal representation of every layer.

Challenges: There is a need to address accuracy compromise resulting from channel pruning. Also, Understanding the impact of channel elimination on different layers' performance remains an open issue.

3.2.2 Filter Pruning

Filter pruning, a technique for compressing convolutional neural networks (CNNs), involves the removal of entire filters from the network. The literature review outlines various techniques for filter pruning, each with its unique approach:

1. ThiNet[29]: Incorporates filter-level pruning, discarding unimportant filters based on statistical information from the next layer. Establishes filter-level pruning as an optimization problem and employs a greedy algorithm for its solution.

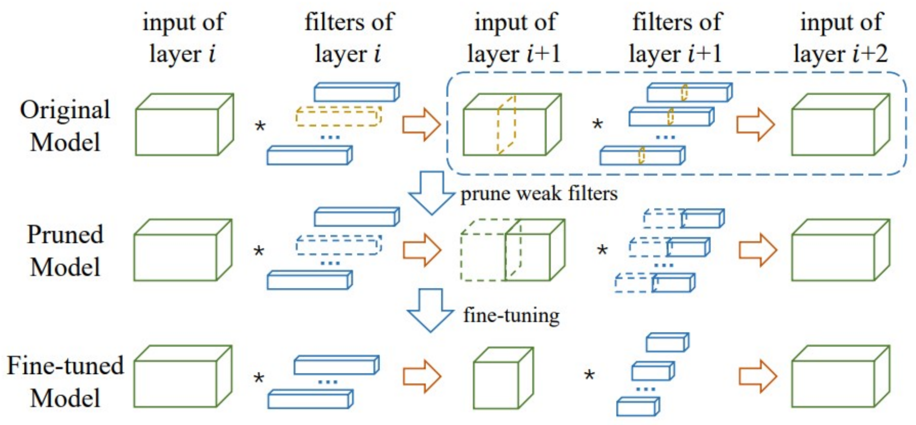


FIGURE 3.2: Visualization of ThiNet: Initially, the attention is directed towards the section enclosed by the dotted box, where a number of less influential channels and their corresponding filters are identified (emphasized in yellow in the initial row). Since these channels (along with their associated filters) make minimal contributions to the overall performance, they can be removed, resulting in a more streamlined model. Subsequently, the network undergoes fine-tuning to restore its accuracy. [29]

2. Sparsification in Fully Connected Layer and Convolutional Filters [1]: Removes sparsification in fully connected layers and convolutional filters to reduce storage requirements in both training and inference phases. Hypothesis: Increasing computational and space complexity of DNN models using sparsification of layers and convolutional filters.
3. Multiobjective Optimization Problem Followed by Knee-Guided Evolutionary Algorithm[49]: Defines filter pruning as a multiobjective optimization problem followed by a knee-guided evolutionary algorithm. It establishes a trade-off between the number of parameters and performance compromise, aiming to prune parameters with minimal performance degradation.
4. Low-Rank Approximation[8]: Speeds up the test-time evaluation of large DNN models designed for object recognition tasks. Utilizes redundancy within convolutional filters to derive approximations that significantly reduce required computation. Starts compression by finding a suitable low-rank approximation for each convolutional layer and fine-tunes until prediction performance is restored.

Challenges: Determining the optimal number of filters to prune remains a challenge. Understanding the effect of filter pruning on the performance of different layers in the DNN model is an open issue.

3.2.3 Connection Pruning

Connection pruning, a technique for compressing neural networks, involves the removal of entire connections from the network.

1. Learning Only Important Connections Using Dropout[15]: Prunes the DNN model by learning only important connections using dropout. Learned weights are quantized using clustering, followed by Huffman coding for reduced storage requirements.
2. Energy-Efficient Inference Engine (EIE) [14]: An EIE performs DNN-based inference on the compressed network obtained after pruning redundant connections and weight sharing. EIE accelerates sparse matrix multiplication through weight sharing without sacrificing model accuracy.
3. Sparse CNN (SCNN)[34]: Employs zero-valued weight estimation during training generated from ReLU operation. SCNN enhances performance and energy efficiency using both weight and activation sparsity. Maintains sparse weights and activations in DNN compression, reducing data transmission and storage requirements.
4. Sparse Variational Dropout [33]: An extension of variational dropout considering all possible dropout values for sparse DNN model compression. Incorporates KL-divergence in variational dropout, achieving higher sparsity in CNN and fully connected layers.

$$L(\varphi) = L_D(\varphi) - D_{KL}(r_\varphi(\theta) \,||\, p(\theta|D))$$
$$L_D(\varphi) = \sum_{i=1}^{N} \mathbb{E}_{r_\varphi(\theta)}[\log p(y_i|x_i, \theta)]$$

where: $L(\varphi)$ is the overall loss function, $L_D(\varphi)$ is the data loss function, $D_{KL}(r_\varphi(\theta) \,||\, p(\theta|D))$ is the KL divergence, $r_\varphi(\theta)$ is the variational distribution, $p(\theta|D)$ is the true posterior distribution, y_i is the label for the i-th training instance, x_i is the i-th training instance, θ is the model parameter, N is the number of training instances. The goal is to find the variational distribution $r_\varphi(\theta)$ that minimizes the overall loss function $L(\varphi)$.

5. Bayesian Compression for Deep Learning [28]: Introduces hierarchical priority to prune nodes, incorporating posterior uncertainties for fixed point precision determination. Induces sparsity for hidden neurons, pruning despite individual weights.
6. Splicing[12]: A network compression technique incorporating splicing to avoid incorrect pruning. Solves irretrievable network damage and inconsistency due to inefficient machine by continuous pruning and splicing.

Challenges: Determining the optimal number of connections to prune is a challenge. Understanding the effect of connection pruning on the performance of different layers in the DNN model remains an open issue.

3.2.4 Layer Pruning

Layer pruning, a technique for compressing deep neural networks (DNNs), involves the removal of entire layers from the network.

1. Multi-Tasking Zipping (MTZ)[16]: Merges multiple correlated DNN models for cross-model compression. Utilizes layer-wise neuron sharing to reduce the number of layers in the network.
2. FINN (FINN, 2018) [37]: Builds a fast and flexible heterogeneous architecture for deploying DNNs on embedded devices. Utilizes optimization techniques for mapping binarized neural networks to hardware.
3. Singular Value Decomposition (SVD)-Based Factorization[2]: Factorizes the weight matrix of a layer into three sub-matrices, allowing for the removal of unimportant layers. Specifically designed for the deployment of DNN models on embedded devices.

Challenges: Determining the optimal number of layers to prune remains a challenge. Understanding the effect of layer pruning on the performance of different layers in the DNN model is an open issue.

3.3 Sparse Representation

In contrast to network pruning techniques, sparse representation techniques focus on compressing DNN models while maintaining their overall structure. This approach leverages sparsity in DNN weight matrices, reducing both storage and processing requirements. The sparsity is attributed to two main factors:

1. Zero or Near-Zero Weights: Many weights in the DNN weight matrices are either zero or very close to zero. Removal of these weights leads to a significant reduction in computational and storage overhead.
2. Similar Weight Values: Many stored weights exhibit similarity with each other. This similarity allows for replacing multiple weights with a single weight having multiple connections, further reducing storage and computation requirements.

3.3.1 Quantization

Quantization is a technique for reducing the storage and computation requirements of deep neural networks (DNNs) by decreasing the precision of weights and activations.

1. Post-training Quantization[15]: Quantizes weights and activations of a pre-trained DNN model. Simple and common, but may lead to significant accuracy loss.
2. Quantized Neural Networks (QNNs)[22]: Trains DNN models with quantized weights and activations from scratch. Optimized quantization scheme for a specific DNN model, higher accuracy but requires more training time.
3. Learned Weight Quantization [23]: Learns a quantization scheme for weights and activations during training. Adapts quantization scheme to a specific DNN model and data, balancing accuracy and compression.

Challenges: Reducing accuracy loss and improving the complexity of estimating quantization parameters is an area for future development

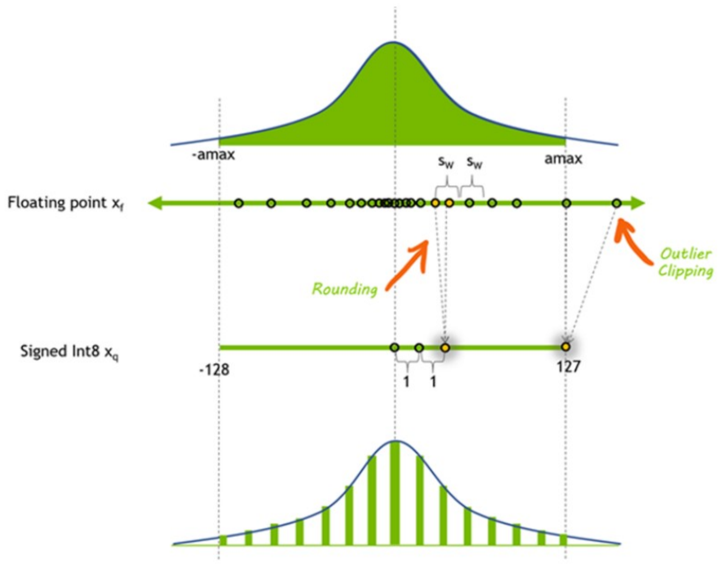


FIGURE 3.3: Mapping of weights in Quantization

3.3.2 Multiplexing

Multiplexing is a technique for reducing the storage and computation requirements of DNNs by sharing weights between different connections. Here's a review of multiplexing techniques:

1. Weight Sharing with Equivalence Partitioning (EIE)[13]: Shares weights between connections with similar activations. Effective for reducing parameters but may lead to accuracy loss.
2. Multi-Tasking Zipping (MTZ) [5]: Shares weights between connections used for different tasks. Effective for DNNs used in multiple tasks but may lead to accuracy loss.
3. Pruning and Splicing [14]: Prunes unimportant connections and splices remaining connections to share weights. Achieves high accuracy but can be complex to implement.
4. Multi-Branch CNN Architecture [47]: Multiplexes different recognition and prediction tasks simultaneously using a single backbone network. Achieves high accuracy and efficiency but can be complex to design and train.

Challenges: Reducing accuracy loss due to multiplexing and improving the complexity of designing and training multiplexed DNNs is a big challenge

3.3.3 Weight Sharing

Weight sharing is a technique for reducing the storage and computation requirements of DNNs by reusing the same weights for multiple connections. Here's a review of weight sharing techniques:

1. K-Means Clustering for Weight Sharing [15]: Uses k-means clustering to identify weights to be shared. Simple and effective, achieves significant reductions in storage and computation without sacrificing accuracy.
2. Sharing of Training Data and Weights[40]: Involves sharing training data, improving accuracy, but raising privacy concerns. Discloses a few samples of training data using a max-margin approach to reduce disclosure.
3. Multi-Tasking Learning for Weight Sharing[11]: Uses multi-tasking learning, training the DNN model on multiple tasks simultaneously. Improves accuracy by optimizing different losses in a multi-tasking framework.

3.4 Bits Precision

Bits precision is a technique aimed at diminishing the storage and computation demands of deep neural networks (DNNs) by decreasing the number of bits used to represent weights and activations. This reduction in the precision of data representation substantially reduces the size of the DNN model, facilitating storage and deployment on embedded devices. However, the trade-off is that bits precision may result in a loss of accuracy, as it involves discarding some information about the weights and activations.

3.4.1 Estimation using Integer Precision

Neuro.ZERO[24] introduces mechanisms for accelerating DNN inference emphasizing runtime adaptations for increased accuracy. Methods:

- **Extended Inference:** Enhances DNN structure for improved accuracy using additional resources from a secondary processor.
- **Expedited Inference:** Speeds up inference by offloading a portion of the task to the secondary processor.
- **Ensemble Inference:** Runs multiple DNN models on both primary and secondary processors simultaneously, combining their outputs.
- **Latent Training:** Primary model runs as usual, but training for unseen data is performed on secondary processors.

3.4.2 Low Bits Representation

Quantized Neural Networks (QNNs)[19]: Operates with weights and activations quantized to very low precision levels, such as 1 bit or 4 bits. Linear Quantization (LQ) involves rounding real-valued numbers within a specified bit length, offering control over precision-accuracy trade-off. Logarithmic Quantization (LogQ) utilizes logarithmic quantization for further reduction in memory requirements and computational efficiency.

3.4.3 Binarization

Binary Neural Networks (BNNs)[20]: Introduces a training method incorporating binary activations and weights. Estimates gradients during model training using binary activations and weights. Binarization Techniques:

- **Deterministic Binarization:** Transforms real-valued numbers into binary values using the Sign($\cdot$) function.
- **Stochastic Binarization:** Introduces a probability for converting real values to binary values, allowing for probabilistic binarization.

Frameworks:

- **FINN**[38]: A fast and flexible DNN compression mechanism, providing a parametric architecture for dataflow and optimized classification on limited-resource devices.
- **cDeepArch**[43]: A compact DNN architecture dividing tasks into subtasks to efficiently utilize limited storage and processing capacity, incorporating binarization for computational reduction.

3.5 Knowledge Distillation

Knowledge distillation is a powerful technique for compressing deep neural networks (DNNs) by transferring knowledge from a larger, more accurate DNN model (teacher) to a smaller, more compressed DNN model (student). This process involves training the student model on the output logits or soft targets of the teacher model, resulting in a more accurate compressed model without significantly increasing its size.

3.5.1 Logits Transfer in Knowledge Distillation

Logits transfer is a knowledge distillation technique wherein the unnormalized output logits of a teacher model serve as soft targets during the training of a student model. The goal is to minimize the disparity between the logits of the student and teacher models, facilitating the transfer of knowledge from a larger, more accurate model to a smaller, compressed one. Some techniques are discussed as follows:

1. **Temperature Scaling** [4]: Soften teacher's logits using a temperature factor. Prevents overfitting of the student model to the teacher's logits, encouraging more generalizable feature learning.
2. **Increasing Neural Network Accuracy by Regulating Wrong Predictions**[46]: Uses logits transfer to distill knowledge from a single DNN model, including its wrong predictions. Improves the accuracy of the student model by using self-knowledge distillation.

3. **RONA**: A Privacy-Preserving DNN Compression Framework[39]: Uses logits transfer to compress a DNN model while preserving privacy. Achieves significant compression and privacy preservation.
4. **Knowledge Adaptation**[6]: Dynamically adjusts the student model's loss function based on the difficulty of training samples. Enables the student model to focus on challenging samples, enhancing knowledge transfer.

3.5.2 Teacher Assistant in Knowledge Distillation

The teacher assistant (TA) approach enhances knowledge distillation by introducing an intermediate layer between the teacher and student models. This intermediary, the teacher assistant, is a smaller model trained on the output logits of the teacher model. Subsequently, the student model is trained on the TA's output logits, which are more akin to its own logits than those of the teacher model.

Benefits:

- **Reduced Gap Between Teacher and Student:** The TA bridges the gap between the teacher and student models, fostering enhanced knowledge transfer and improved accuracy for the student model.
- **Improved Efficiency:** The TA can be designed to be more efficient than the teacher model, contributing to an overall performance boost in the knowledge distillation process.
- **Potential for Higher Order Compression:** According to [32], inserting multiple TAs between the teacher and student models can achieve higher compression ratios without sacrificing accuracy.

Challenges:
Training time for knowledge distillation is prolonged due to the necessity of training the TA. This drawback is particularly significant for large or complex models.

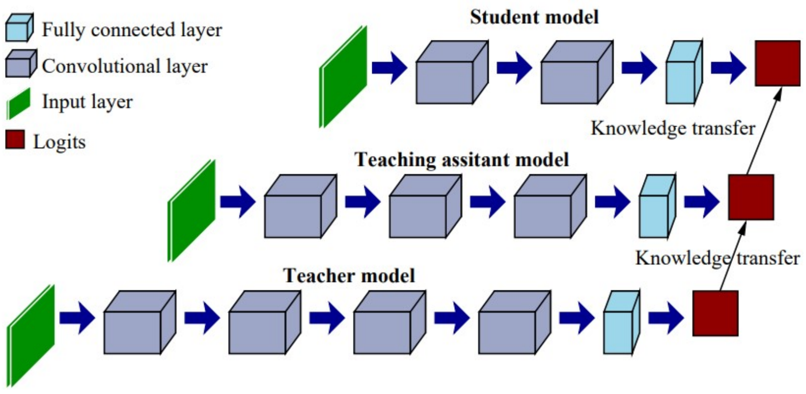


FIGURE 3.4: TA fills the gap between student & teacher

3.5.3 Domain Adaptation Techniques

1. **ShrinkTeaNet**[9]: Shrinking the Gap Between Teacher and Student Models. Introduces angular distillation loss to determine feature direction despite hard targets that can cause overfitting. Angular loss is defined as:

$$L_{adl}(S,T) = -\sum t[i] \cdot \cos(\theta_{t[i]}, \theta_{s[i]})$$

   where $\cos(\cdot)$ estimates the distance between transformed features of teacher ($trans(F_t)$) and student ($trans(F_s)$). Suggests parallel training of student and teacher models to provide soft targets at each step. Soft targets differ from final class label predictions acting as hard targets during training.

2. **MobileDA**[42]: Mobile Domain Adaptation for IoT Devices. Aims to learn transferable features while maintaining a simplified DNN model structure. Cross-Domain Distillation: Trains a student model on IoT devices using a complex teacher model on a high-end machine. Ensures the simplified DNN model handles domain shift during testing on embedded devices. Simultaneously optimizes different loss functions to tackle domain adaptation.

3.6 Miscellaneous

Diverse approaches in the miscellaneous category, such as DeepSense[44], FastDeepIoT[45], AdaDeep[26], NestDNN[10], DeepEye[31], DeepApp[36], DeepFusion[41], and ShuffleNet[48], effectively reduce storage and computation needs.

Chapter 4

Methodology

4.1 Software Implementation

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

1. First, we **prune** the networking by removing the redundant connections, keeping only the most informative connections.
2. 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.
3. Finally, we apply **Huffman coding** to take advantage of the biased distribution of effective weights.

4.1.1 Pruning

We commence by establishing connectivity through standard network training. Subsequently, we undertake pruning by removing connections with weights below a defined threshold.

The sparse structure resulting from pruning is stored using compressed sparse row (CSR) or compressed sparse column (CSC) format, necessitating $2a+n+1$ numbers, where ’a’ represents the number of non-zero elements, and ’n’ is the number of rows or columns.

To achieve further compression, we employ the storage of index differences instead of absolute positions. These differences are encoded in 8 bits for convolutional layers and 5 bits for fully connected layers. In instances where the index difference exceeds the specified bound, we implement a zero-padding solution: for differences exceeding 8, equivalent to the largest 3-bit unsigned number, a filler zero is added.

4.1.2 Quantization and Weight Sharing

Through weight sharing and network quantization, the pruned network can be further compressed by lowering the number of bits needed to represent each weight. By allowing many connections to share the same weight and then fine-tuning those shared weights, we reduce the number of effective weights we need to store.

The goal of trained quantization is to reduce the precision of the weights in the network. This is done by quantizing the weights to a smaller number of bits, ensuring that the network does not lose accuracy.

The trained quantization process works as follows:

1. The network is first pruned to remove unimportant connections.
2. The weights in the pruned network are then quantized to a smaller number of bits.
3. The quantized network is then trained on the training data.
4. The trained quantized network is evaluated on the test data.

Given k clusters, we only require $\log_2(k)$ bits to encode the index, allowing us to calculate the compression rate. If n connections make up the network and each connection is represented by b bits, then limiting the connections to have a maximum of k shared weights will yield the following compression rate:

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

4.1.2.1 K-Means Clustering

To reduce storage requirements, we employ k-means clustering to group similar weights within each layer of the trained network. These centroids of the one-dimensional k-means clustering serve as the shared weights. This approach does not share weights across layers.

The WCSS is the sum of the squared distances between all of the weights in a cluster and its centroid. By minimizing the WCSS, the k-means clustering algorithm ensures that the weights within each cluster are as similar as possible.

$$\arg\min_{C} \sum_{k=1}^{K} \sum_{w \in c_k} ||w - c_k||^2 \tag{4.1}$$

where:

C is the set of k clusters

c_k is the centroid of cluster k

w is a weight in the network

$||w - c_k||^2$ is the squared distance between w and c_k

This 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 gradient for each shared weight is calculated and applied to update the shared weight during back-propagation. The gradient of the centroids is computed as follows:

$$\frac{\partial L}{\partial C_k} = \sum_{i,j} \frac{\partial L}{\partial W_{ij}} 1(I_{ij} = k) \tag{4.2}$$

where,

L is the loss

W_{ij} is the weight in the ith column and jth row

I_{ij} is the centroid index of element W_{ij}

C_k is the kth centroid of the layer

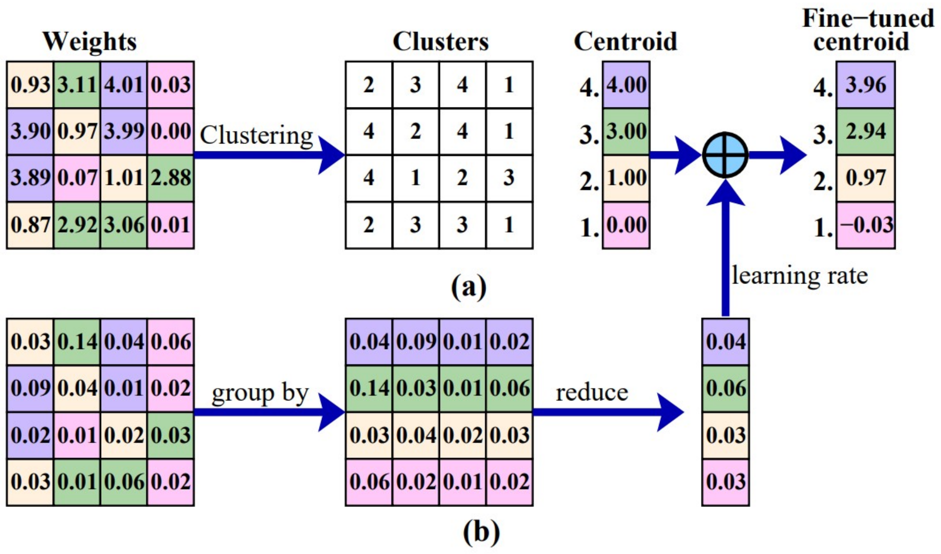


Figure 4.1: Illustration of weight quantization using clustering Part (a) Forward propagation and part (b) backward propagation

4.1.3 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. Previous experiments have shown that Huffman coding can reduce network storage by 20-30%.

4.2 Hardware Implementation of Fully Connected Layer

4.2.1 Neuron Architecture

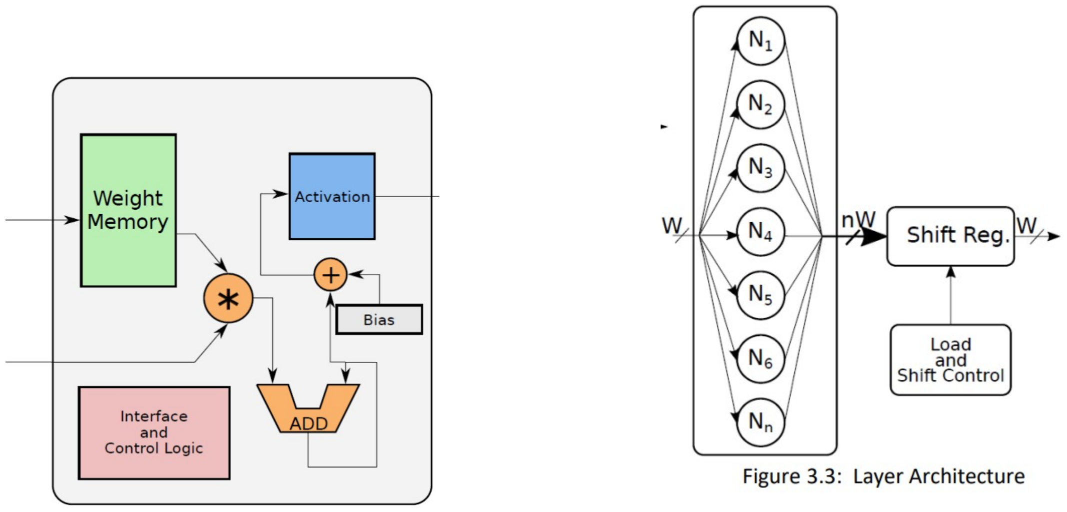
ZyNet's artificial neuron architecture features independent interfaces for configuration and data. Each neuron, regardless of predecessor count, has a single data input interface, promoting network scalability and optimizing clock performance at the expense of latency.

Internal memory, 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.

The MAC unit output is processed by an activation unit, implementing a look-up-table-based (LUT) or circuit-based function based on the configured activation function (e.g., Sigmoid, ReLU). The choice impacts network accuracy, resource utilization, and clock performance. Optionally, the user can specify the depth of the Sigmoid function's Look-Up Table, or it can be automatically determined by the tool.

4.2.2 Layer Architecture

Each layer creates a user-specified number of neurons and oversees data flow between layers. As each neuron features a single data interface and a fully connected layer necessitates connections to every neuron from the preceding layer, data from each layer is initially stored in a shift register. This data is subsequently shifted to the next layer, one per clock cycle, as illustrated in Fig. 3.3. The tool automatically implements connections between layers and integrates input and output AXI interfaces.



(a) (b)

FIGURE 4.2: (a) Neuron Architecture (b) Layer Architecture

4.3 Hardware Implementation of Convolution

The convolution operation is implemented differently in hardware than in software. This hardware implementation is written in Verilog, a hardware description language, using the Vivado Suite from Xilinx. The simulation was performed using Vivado with the target board set as a ZYNQ FPGA board. The implementation is broken down into modules, each of which performs a specific step in the convolution operation.

To perform the convolution process, the structure has four modules that are implemented in Vivado:

1. Line Buffer
2. Multiply and Accumulate Unit
3. Control Unit
4. Top Module

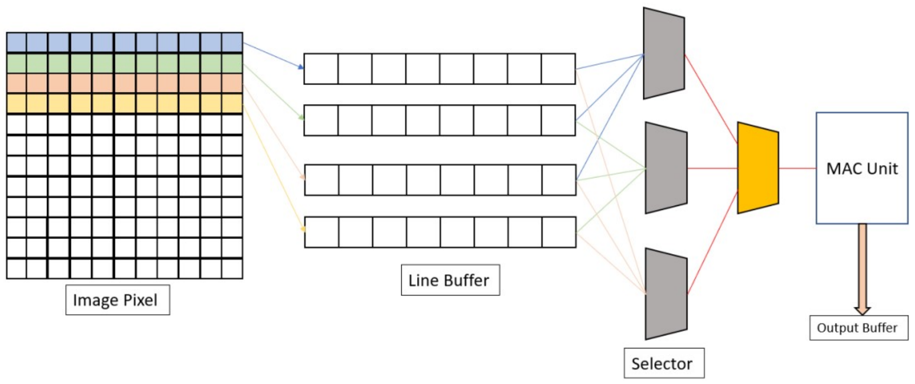


FIGURE 4.3: Flow Diagram of Convolution Circuit

4.3.1 Line Buffer

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. The buffer's output size is determined by the implemented Kernel size, ensuring a continuous bit stream. Despite the apparent continuity, the internal structure relies on registers to orderly store color values and the necessary pixels in a row. In this specific setup, predetermined parameters include: Image Width: 512 Image Color Encoding: 8 Bits Kernel Size: 3×3 Resulting in an output size of 24 bits. $Output = 3 \times ColorEncoding$

The additional inputs consist of the subsequent signals:

- Input Clock Signal: Facilitates synchronization for line buffer operation.
- Reset Signal: Initiates a reset of the buffer state to 0 at the start.
- Input Data Valid Signal: Functions as a flag signal, enabling the line buffer to initiate reading from Image Pixel Data.
- Read Data Signal: Governs the output control of the line buffer.

4.3.2 Multiply and Accumulate

In computer architecture, particularly in digital signal processing, the multiply–accumulate (MAC) operation computes the product of two numbers and adds it to an accumulator using a hardware unit known as a multiplier–accumulator (MAC unit). In

Convolutional Neural Network (CNN) operations, a MAC unit is employed. Using a Hardware Description Language for MAC operations allows simultaneous multiplication and addition, enhancing speed. The MAC unit's speed determines the convolution operation speed, offering flexibility for algorithm experimentation. The multiplier and adder width depends on the kernel size; a 3×3 kernel in this implementation requires 9 parallel data paths.

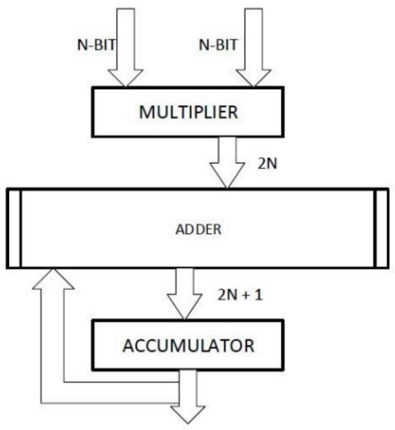


FIGURE 4.4: A typical structural description of a MAC unit

The inputs and outputs for this MAC module are outlined as follows:

- Input Clock Signal: Synchronizes the MAC operation.
- Input Pixel Data: A continuous 72-bit pixel data stream representing nine image pixels, distributed to various data paths as needed.
- Input Data Valid Signal: A flag signal authorizing the commencement of the MAC unit's operation.
- Output Convolved Data: A single 8-bit pixel value computed at the conclusion of the MAC operation.
- Output Convolved Data Valid Signal: A flag signal indicating to the Output Buffer when to capture the output.

4.3.3 Control Unit

This module oversees the operation of the line buffers, ensuring the synchronous operation of the circuit. The input/outputs for the control module are detailed as follows:

- Input Clock Signal: Synchronizes MAC operation.
- Input Reset Signal: External signal asserted before initiating the operation.
- Input Pixel Data: Flow of data from image pixels to the line buffer.
- Output Pixel Data: Data flowing from the line buffers to the MAC unit.
- Output Pixel Data Valid: Flag signal indicating to the MAC unit when to operate on the output.
- Output Interrupt: Determines the order of line buffer write operations.

The control module maintains the sequence in which line buffers are filled. Initially, all three line buffers must be filled before initiating MAC unit operation. Concurrently, the fourth line buffer is filled to leverage parallelism. As the convolution operation for a single row concludes, the control module dispatches the next set of three line buffers while the first one is refilled. This is accomplished through the generation of select signals and output signals to prompt the MAC unit to commence operation.

4.3.4 Top Module and Test Bench

The top module is defined primarily for simulation purposes and is also utilized during the IP packaging process for deployment on the FPGA. It incorporates instances for the Control Module, MAC Unit, and Output Buffers. The Line Buffers are instantiated within the Control Module itself. A test bench is crafted where the Top Module serves as the device under test (DUT).

The test bench reads a greyscale image file in BMP format with dimensions 512×512. It extracts the binary data by excluding the header information and subsequently feeds this data to the Device Under Test (DUT). The DUT's output is then retrieved from the buffer and written back into a BMP file, preserving the original header information.

Chapter 5

Experimental Results

5.1 Software Implementation

The optimizations defined earlier were implemented on a LeNet network and by training and evaluating on the MNIST data set[7]. 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

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)
```

Listing 5.2: After Pruning

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

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

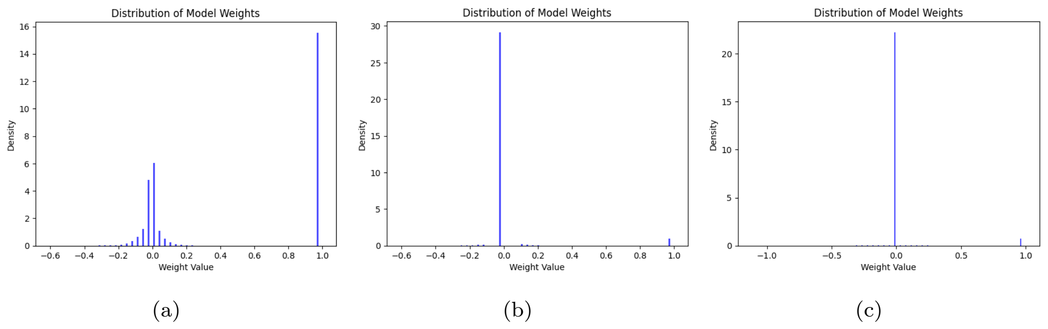


Figure 5.1: Distribution of weights: (a) Before Pruning (b) After Pruning (c) Retraining after Pruning

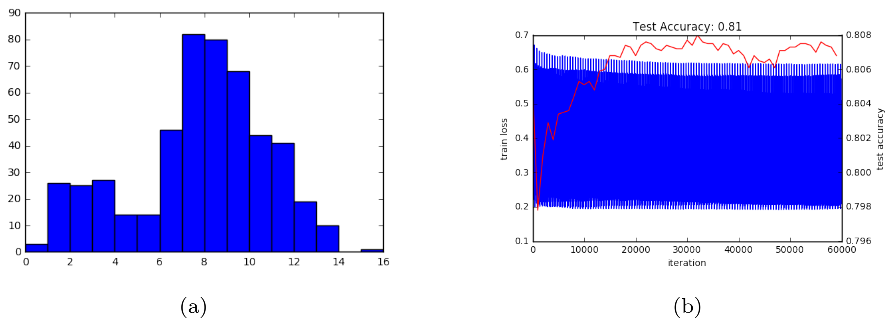


Figure 5.2: Model trained on CIFAR10 dataset. Number of Clusters set to 16: (a) Weight distribution (b) Accuracy and Loss Plot

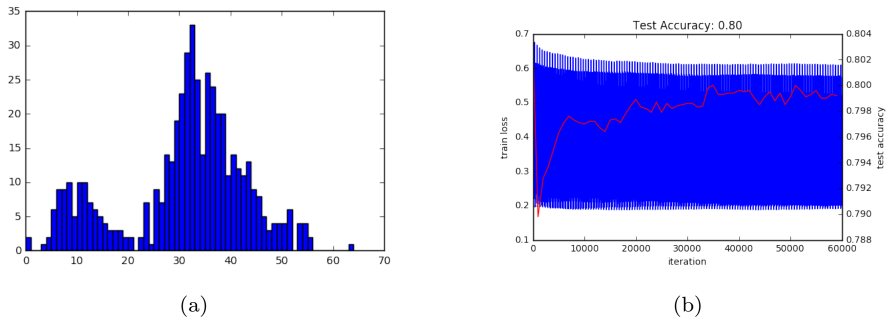


Figure 5.3: Model trained on CIFAR10 dataset. Number of Clusters set to 64: (a)Weight distribution (b) Accuracy and Loss Plot

5.2 Hardware Implementation

In this section we discuss the implementation results and performance of ZyNet based DNNs. All implementations adhere to a 5-layer architecture comprising 784 neurons in the input layer, two hidden layers with 30 neurons each, one hidden layer with 10 neurons, and an output layer with 10 neurons. The output layer is linked to a hardmax module for identifying the neuron with the maximum output value. All designs undergo simulation and implementation using Xilinx Vivado 2018.3, and hardware validation is performed on a ZedBoard equipped with an xc7z020clg484-1 SoC and 512MB external DDR3 memory.

5.2.1 Resource Utilization and Timing analysis

For the given results, the input values have a datawidth of 16. We'll compare the Resource Utilization, Maximum Frequency, and Power Consumption of the Zedboard for a single neuron using the following graphs.

The overall resource utilization on the board is as follows: LUTs: 53,200, FFs: 106,400, BRAM: 140, DSP slices: 220. In the case of ReLU activation function implementation, it consumes 68 LUTs, 81 FFs, 0.5 BRAM, and 2 DSP slices.

5.2.2 Convolution

By employing parallelism and utilizing the data from three line buffers for MAC operation while simultaneously filling the fourth line buffer with new data, we operate with a total of four line buffers. However, if we were to reduce the total number of line buffers to three, we would need to wait for the data to be filled in the line buffer before performing the MAC operation. This sequential process increases the overall time required to perform convolution by two times, as it introduces additional waiting time for data filling before MAC operation can commence. Therefore, it can be inferred that increasing the number of line buffers or expanding the number of hardware components can enhance parallelism and subsequently reduce computation time.

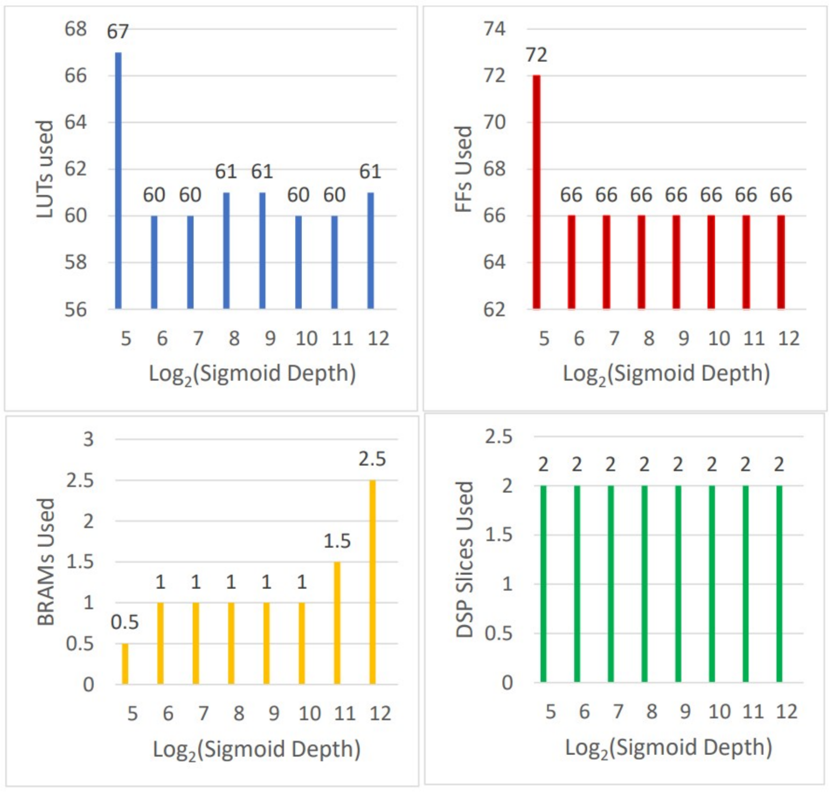


FIGURE 5.4: Resource Utilisation for Sigmoid Function with varying Sigmoid depths

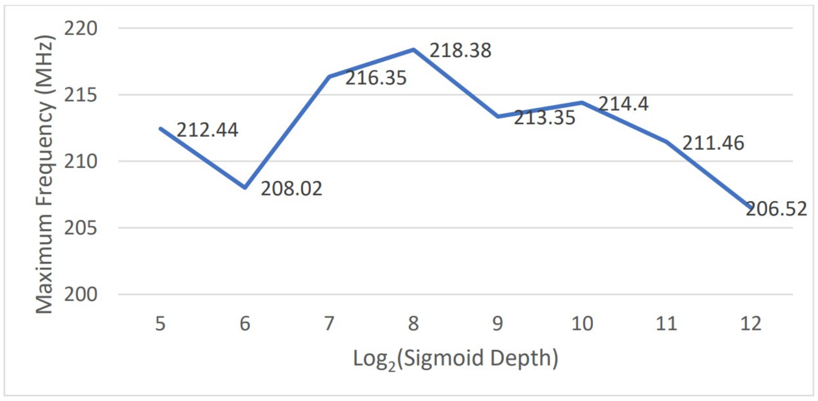


FIGURE 5.5: Maximum Frequency for Sigmoid Function with varying Sigmoid depths

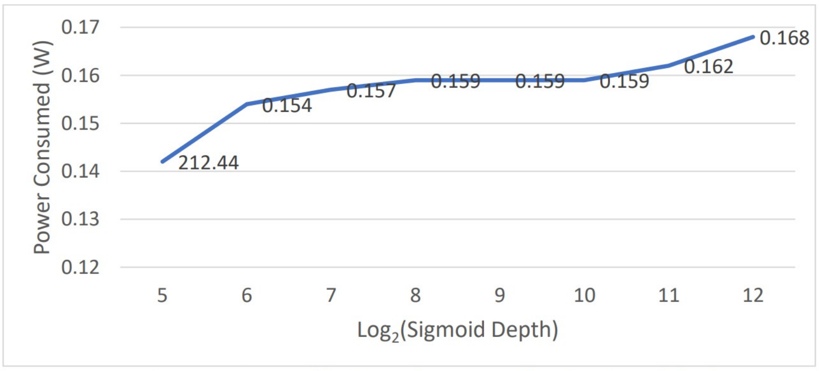
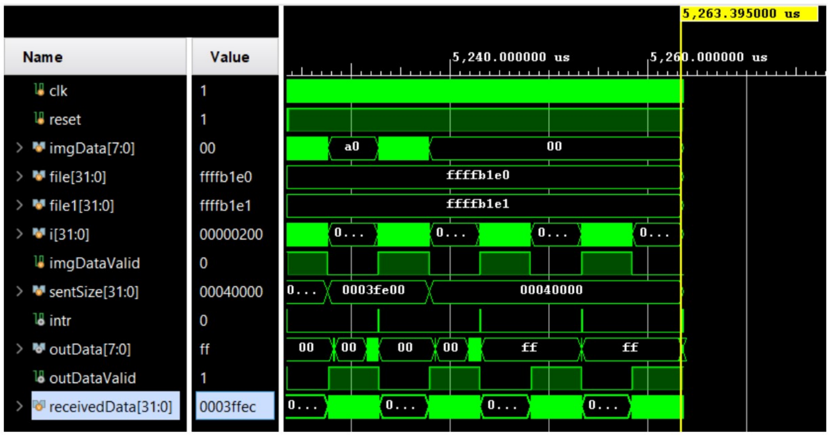
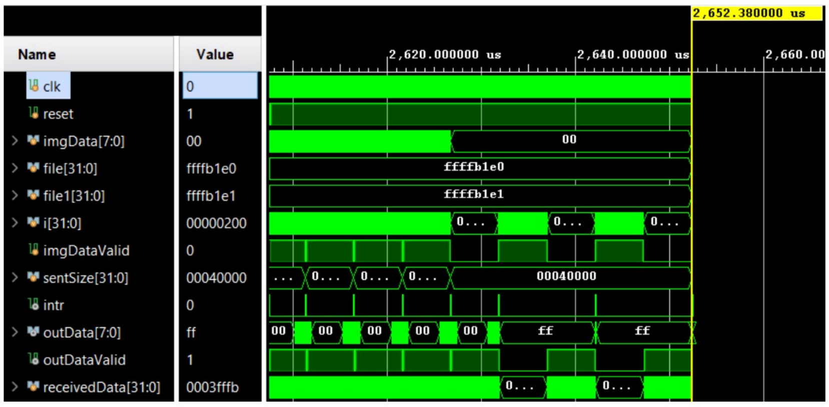


FIGURE 5.6: Power Consumed for Sigmoid Function with varying Sigmoid depths



Name	Value
clk	1
reset	1
imgData[7:0]	00
file[31:0]	ffffb1e0
file1[31:0]	ffffb1e1
i[31:0]	00000200
imgDataValid	0
sentSize[31:0]	00040000
intr	0
outData[7:0]	ff
outDataValid	1
receivedData[31:0]	0003ffec

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



Name	Value
clk	0
reset	1
imgData[7:0]	00
file[31:0]	ffffb1e0
file1[31:0]	ffffb1e1
i[31:0]	00000200
imgDataValid	0
sentSize[31:0]	00040000
intr	0
outData[7:0]	ff
outDataValid	1
receivedData[31:0]	0003fffb

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

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In conclusion, our exploration delved into the fundamental concepts of Convolutional Neural Networks (CNNs) and the imperative motivation behind compressing these networks. The investigation extended to a thorough analysis of various optimization techniques aimed at facilitating swifter inference in neural networks. Our practical implementation involved the application of optimization techniques in software, executed on Google Colab, and the subsequent evaluation of model performance on CIFAR-10 and MNIST datasets. Remarkably, our findings revealed that neural network compression did not significantly compromise accuracy.

Transitioning to the realm of hardware implementation, we embarked on designing and implementing the convolution operation at the circuit level using Verilog HDL. This circuit-level approach prompted a paradigm shift in our understanding of the operation, as evidenced by the detailed discussions on module design and simulations presented in this report. The hardware implementation enabled substantial gains in operational efficiency through thoughtful pipelining and parallelization of the convolution operation.

In essence, our comprehensive exploration—from software-based optimization to circuit-level hardware implementation—offers valuable insights into the intricacies of accelerating CNN inference on FPGA platforms. As the demand for efficient neural network deployment on custom hardware accelerators continues to rise, our work contributes to the ongoing discourse on achieving rapid and resource-efficient inference through a harmonious integration of hardware and software approaches.

6.2 Future Work

The project lays the groundwork for several promising future directions:

1. 3D Convolution Extension: Exploring the extension of 2D convolution operations to 3D offers opportunities to implement neural networks on volumetric data, expanding application possibilities.
2. Complex CNN Model Implementation: The current hardware acceleration and optimization techniques provide a foundation for implementing complete CNN models like AlexNet, VGGNet, ResNet, etc., on FPGA. Resource constraints may require further optimization or alternative hardware solutions.
3. C to Verilog Code Translation: Utilizing C programming, followed by translation into Verilog using Xilinx Vivado tools, enables a seamless transition between high-level programming and hardware description.
4. Ongoing Neural Network Implementation in C: The ongoing work on implementing neural networks in C, especially the conversion of LeNet to Verilog, reflects a progressive approach to streamline development processes.
5. Advanced Hardware Optimization Techniques: 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 performance.

In summary, future work includes exploring 3D convolutions, extending the implementation of complex CNN models, investigating C-to-Verilog code conversion, advancing ongoing neural network implementation in C, and refining optimization techniques for hardware deployment. These directions contribute to the ongoing evolution of efficient and scalable neural network implementations on FPGA platforms.

Bibliography

[1] Sourav Bhattacharya and Nicholas Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. pages 176–189, 11 2016.

[2] J. Chauhan, J. Rajasegaran, S. Seneviratne, A. Misra, A. Seneviratne, and Y. Lee. Performance characterization of deep learning models for breathing-based authentication on resource-constrained devices. *Proc. IMWUT*, 2(4):1–24, 2018.

[3] S. Chen, L. Lin, Z. Zhang, and M. Gen. Evolutionary netarchitecture search for deep neural networks pruning. In *Proc. ACAI*, pages 189–196, 2019.

[4] Z. Chen, L. Zhang, Z. Cao, and J. Guo. Distilling the knowledge from hand-crafted features for human activity recognition. *IEEE Transactions on Industrial Informatics*, 14(10):4334–4342, 2018.

[5] Z. Chen, T. Zhang, M. Sun, and Y. Yao. Multi-tasking learning with low-rank factorization. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1230–1239, 2017.

[6] X. Cheng, Z. Rao, Y. Chen, and Q. Zhang. Explaining knowledge distillation by quantifying the knowledge. In *Proc. CVPR*, pages 12 925–12 935, 2020.

[7] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[8] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Proc. NIPS*, pages 1269–1277, 2014.

[9] C. N. Duong, K. Luu, K. G. Quach, and N. Le. Shrinkteanet: Million-scale lightweight face recognition via shrinking teacher-student networks. *arXiv preprint arXiv:1905.10620*, 2019.

[10] B. Fang, X. Zeng, and M. Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proc. MobiCom*, pages 115–127, 2018.

[11] P. Georgiev, S. Bhattacharya, N. D. Lane, and C. Mascolo. Low-resource multi-task audio sensing for mobile and embedded devices via shared deep neural network representations. *Proc. IMWUT*, 1(3):1–19, 2017.

[12] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. In *Proc. NIPS*, pages 1379–1387, 2016.

[13] S. Han, J. Kang, H. Mao, Y. Hu, and X. Li. Eie: Efficient inference engine for deep learning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 218–231, 2017.

[14] 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. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[15] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[16] X. He, Z. Zhou, and L. Thiele. Multi-task zipping via layer-wise neuron sharing. In *Proc. NIPS*, pages 6016–6026, 2018.

[17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, 2017.

[18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[20] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proc. NIPS*, pages 4107–4115, 2016.

[21] Fatima Hussain, Rasheed Hussain, Syed Ali Hassan, and Ekram Hossain. Machine learning in iot security: Current solutions and future challenges. *IEEE Communications Surveys & Tutorials*, 22(3):1686–1721, 2020.

[22] 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 *Proc. CVPR*, pages 2704–2713, 2018.

[23] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang. Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors. In *Proc. DAC*, pages 1–6, 2018.

[24] S. Lee and S. Nirjon. Neuro. zero: A zero-energy neural network accelerator for embedded sensing and inference systems. In *Proc. SenSys*, pages 138–152, 2019.

[25] C. Liu and Q. Liu. Improvement of pruning method for convolution neural network compression. In *ICDLT*, pages 57–60, 2018.

[26] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proc. MobiSys*, pages 389–400, 2018.

[27] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proc. ICCV*, pages 2736–2744, 2017.

[28] C. Louizos, K. Ullrich, and M. Welling. Bayesian compression for deep learning. In *Proc. NIPS*, pages 3288–3298, 2017.

[29] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *CoRR*, abs/1707.06342, 2017.

[30] Vicent Sanz Marco, Ben Taylor, Zheng Wang, and Yehia Elkhatib. Optimizing deep learning inference on embedded systems through adaptive model selection. *CoRR*, abs/1911.04946, 2019.

[31] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proc. MobiSys*, pages 68–81, 2017.

[32] S.-I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. Improved knowledge distillation via teacher assistant. *arXiv preprint arXiv:1902.03393*, 2019.

[33] D. Molchanov, A. Ashukha, and D. Vetrov. Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*, 2017.

[34] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[35] Surbhi Saraswat, Hari Prabhat Gupta, and Tanima Dutta. A writing activities monitoring system for preschoolers using a layered computing infrastructure. *IEEE Sensors Journal*, 20(7):3871–3878, 2020.

[36] Z. Shen, K. Yang, W. Du, X. Zhao, and J. Zou. Deepapp: A deep reinforcement learning framework for mobile application usage prediction. In *Proc. SenSys*, pages 153–165, 2019.

[37] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proc. CVPR*, pages 2820–2828, 2019.

[38] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proc. FPGA*, pages 65–74, 2017.

[39] J. Wang, W. Bao, L. Sun, X. Zhu, B. Cao, and S. Y. Philip. Private model compression via knowledge distillation. In *Proc. AAAI*, pages 1190–1197, 2019.

[40] H. Wu, C. Wang, J. Yin, K. Lu, and L. Zhu. Sharing deep neural network models with interpretation. In *Proc. WWW*, pages 177–186, 2018.

[41] H. Xue, W. Jiang, C. Miao, Y. Yuan, F. Ma, X. Ma, Y. Wang, S. Yao, W. Xu, A. Zhang, et al. Deepfusion: A deep learning framework for the fusion of heterogeneous sensory data. In *Proc. MobiHoc*, pages 151–160, 2019.

[42] J. Yang, H. Zou, S. Cao, Z. Chen, and L. Xie. Mobileda: Towards edge domain adaptation. *IEEE Internet of Things Journal*, 2020.

[43] K. Yang, T. Xing, Y. Liu, Z. Li, X. Gong, X. Chen, and D. Fang. Cdeeparch: A compact deep neural network architecture for mobile sensing. *IEEE/ACM Transactions on Networking*, 27(5):2043–2055, 2019.

[44] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proc. WWW*, pages 351–360, 2017.

[45] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proc. SenSys*, pages 278–291, 2018.

[46] S. Yun, J. Park, K. Lee, and J. Shin. Regularizing class-wise predictions via self-knowledge distillation. In *Proc. CVPR*, pages 13 876–13 885, 2020.

[47] J. Zhang, B. Ye, and X. Luo. Mbfn: A multi-branch face network for facial analysis. In *Proc. ACAI*, pages 542–549, 2019.

[48] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proc. CVPR*, pages 6848–6856, 2018.

[49] Y. Zhou, G. G. Yen, and Z. Yi. A knee-guided evolutionary algorithm for compressing deep neural networks. *IEEE Transactions on Cybernetics*, pages 1–13, 2019.