Energy Efficient Hardware Architectures for Memory Prohibitive Deep Neural Networks

vorgelegt von
M.Tech.
Suhas Shivapakash
ORCID: 0000-0002-9173-213X

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
-Dr.-Ing.-

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr.-Ing. Wolfgang Heinrich
Gutachter: Prof. Dr.-Ing. Friedel Gerfers
Gutachter: Prof. Dr.-Ing. Hitesh Srimali
Gutachter: Prof. Dr.-Ing. Samek Wojciech
Tag der wissenschaftlichen Aussprache: 19. Oktober 2023

Berlin 2024

Dedicated to my family, friends and loved ones

"Do your work with honesty, integrity and dedication. Leave the rest to me as the process of working is more important than the actual result"

- Lord Sri Krishna

Acknowledgements

First, I would like to thank Professor Friedel Gerfers for his support and guidance over the past four years. The entire Ph.D. in the chair of Mixed Signal Circuit Design has been a successful and fantastic journey, and I am grateful to be part of it. My Ph.D. would not have been successful if not for his dedication to my projects and my success. He believed in my abilities and gave me the necessary confidence to work on complex projects.

Many contributions in my thesis are due to the collaborations and support from the different groups. First, I would like to thank Hardik Jain and Professor Olaf Hellwich from the Computer Vision and Remote Sensing group for their continuous support in developing the Multi-bit Accelerator; without their support, the project would not have been possible. Similarly I would like to thank Simon Wiedemann, Pablo Widemann, Daniel Becking, Dr. Wojciech Samek and Professor Thomas Wiegand from the Machine Learning Group of Fraunhofer Heinrich Hertz Institute (HHI). Furthermore, their support and guidance played a very important role in the FantastIC4 project.

Thanks to all the Chair of Mixed Signal Circuit Design members who have traveled with me on this Ph.D. journey and shared their knowledge, friendship, and support.

Special thanks to Werner Eschenberg, who had been continuous support during my Ph.D., and my friend Amogha KS who helped me adjust to Berlin.

Thanks to my parents, Sudha Prakash and T.P Shivaprakash, my brother Ullas Shivaprakash, my wife Sanjana Athreya, and my best friends Hardik Jain, Vaibhav Krishna and Megha Jain for their unconditional support. Their moral support was very crucial in helping me achieve my desired goals.

IEEE DISCLAIMER

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of TU Berlin's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Abstract

Deep Neural Networks (DNN) form the backbone of modern Artificial Intelligence (AI) systems. However, due to the high computational complexity and divergent shapes and sizes. Dedicated hardware accelerators are required to achieve very high performance and energy efficiency across various DNNs to enable AI in real-world applications. To address this problem and improve the DNN processor's energy efficiency, we introduce the multi-bit accelerator. The multi-bit accelerator achieves the energy-efficient goals for a low-power DNN processor by truncating the preceding layer's partial sums (PSums) before feeding it as an input to the next layer. Row Stationary (RS) dataflow method is used to implement the design. We start inferencing with the high order bit-width like 32-bits for the first convolution layers and sequentially truncate the bits on the MSB/LSB of the integer and the LSB of the fraction part. Even with the sequential truncation, the processor could achieve a top-1 accuracy of up to 14-bits and top-5 accuracy of up to 10-bits. The proposed truncation scheme helped in reducing the resource utilization by 73.25% for LUTs (Lookup tables), 68.76% for FFs (Flip Flops), and 74.60% for BRAMs (Block RAMs), and 79.425% in DSPs (Digital Signal Processors). The multi-bit accelerator could achieve an overall throughput of 223.39 GOPS on a Virtex Ultra Scale FPGA. The corresponding ASIC version implemented on the GF 22nm FDSOI could achieve an overall throughput of 2.03 TOPS/W with a total power consumption of 791mW and an overall area of 1.2mm $\times$ 1.2mm.

In order to further improvise the energy efficiency and area efficiency, we introduce a hardware-software co-designed FantastIC4 accelerator to handle the compact representations of the fully connected layers and reduce the total number of required multipliers to four. In order to make the DNN models amenable for efficient execution, the DNN models are trained to handle the 4-bit quantization. The FantastIC4 accelerator achieves a very high throughput of 2.45 TOPS due to the extreme compression of the models with an overall power consumption of 3.6W on a Virtex Ultrascale FPGA. The equivalent ASIC version implemented on a GF 22nm FDSOI achieves a very high energy efficiency of 20.17 TOPS/W. When compared to other accelerators designed for Google Speech Command (GSC) dataset, FantastIC4 is better by 51$\times$ in terms of throughput and 145$\times$ in terms of area efficiency ($GOPS/mm^2$).

Zusammenfassung

Tiefe neuronale Netze bilden das Rückgrat moderner Systeme der Künstlichen Intelligenz (KI). Aufgrund der hohen Berechnungskomplexität und der unterschiedlichen Formen und Größen sind jedoch spezielle Hardwarebeschleuniger erforderlich. Dedizierte Hardware Beschleuniger sind erforderlich, um eine sehr hohe Leistung und Energieeffizienz bei verschiedenen DNNs zu erreichen, um eine KI in realen Anwendungen zu ermöglichen. Um dieses Problem zu lösen und die Energieeffizienz des DNN prozessors zu verbessern, führen wir den Multi-Bit Beschleuniger ein. Der Multi-Bit Beschleuniger erreicht die energieeffizienten Ziele für einen stromsparenden DNN Prozessor, indem er die Partialsummen (PSums) der vorhergehenden Schicht abschneidet, bevor er sie als Eingabe für die nächste Schicht verwendet. Zur Implementierung des Entwurfs wird die Zeile Stationär Datenflussmethode verwendet. Wir beginnen die Inferenz mit einer Bitbreite hoher Ordnung wie 32 bits für die ersten Faltungsschichten und schneiden die bits am MSB/LSB der Ganzzahl und am LSB des Bruchteils sequentiell ab. Selbst mit der sequentiellen Abschneidung konnte der Prozessor eine Top-1 Genauigkeit von bis zu 14 bits und eine top-5 genauigkeit von bis zu 10 bits erreichen. Das vorgeschlagene Abschneidungsschema trug dazu bei, die Ressourcennutzung bei LUTs (Lookup-Tabellen) um 73,25%, bei FFs (Flip Flops) um 68,76%, bei BRAMs (Block-RAMs) um 74,60% und bei DSPs (Digitalen Signal Prozessoren) um 79,425% zu reduzieren. Der Multi-Bit Beschleuniger konnte auf einem Virtex Ultra Scale FPGA einen Gesamtdurchsatz von 223,39 GOPS erreichen. Die entsprechende ASIC Version, die auf dem GF 22nm FDSOI implementiert wurde, konnte einen Gesamtdurchsatz von 2,03 TOPS/W bei einer Gesamtleistungsaufnahme von 791mW und einer Gesamtfläche von 1,2mm $\times$ 1,2mm erreichen.

Um die Energie- und Flächeneffizienz weiter zu verbessern, führen wir einen von Hardware und Software gemeinsam entwickelten FantastIC4 Beschleuniger ein, der die kompakten Darstellungen der vollständig verknüpften Schichten verarbeitet und die Gesamtzahl der erforderlichen Multiplikatoren auf vier reduziert. Um die DNN Modelle für eine effiziente Ausführung geeignet zu machen, werden die DNN Modelle für die 4-Bit Quantisierung trainiert. Der FantastIC4 Beschleuniger erreicht aufgrund der extremen Komprimierung der Modelle einen sehr hohen durchsatz von 2,45 TOPS bei einer Gesamtleistungsaufnahme von 3,6 W auf einem Virtex Ultrascale FPGA. Die äquivalente ASIC Version, die auf einem GF 22nm FDSOI implementiert ist, erreicht eine sehr hohe Energieeffizienz von 20.17 TOPS/W. Im Vergleich zu anderen Beschleunigern, die für den Google Speech Command (GSC) Datensatz entwickelt wurden, ist FantastIC4 beim Durchsatz um das 51-fache und bei der Flächeneffizienz um das 145-fache besser ($GOPS/mm^2$).

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence. xv

ANN Artificial Neural Networks. 4

ASIC Application Specific Integrated Circuits. 7

BN Batch Normalization. xv

BRAM Block RAMs. xix

CMOS Complementary Metal Oxide Semiconductor. 1

CNN Convolutional Neural Network. 23

CONV Convolution. xix

DNN Deep Neural Network. xv

DRAM Dynamic Random Access Memory. 5

DSP Digital Signal Processor. xix

FC Fully Connected. 6

FIFO First In First Out. 64

FPGA Field Programmable Gate Arrays. 7

GPU Graphics Processing Unit. 20

Ifmaps Input Feature Maps. 24

IoT Internet of Things. 4

KWS Keyword Spotting. xx

MAC Multiply and Accumulate. xv

ML Machine Learning. 1

MLP Multi Layer Perceptron. 4

MNIST Modified National Institute of Standards and Technology Database. 2

Ofmaps Output Feature Maps. 24

PSums Partial Sums. 6

ReLU Rectified Linear Unit. 14

SoA State of the Art. xx

SRAM Static Random Access Memory. 6

TOPS Terra Operations Per Second. 7

1 Introduction

Due to the abundance of audio, video, image, and text data, AI and related fields have developed tremendously in the past decade. There is aggressive research in AI, its techniques, inference, and applications in these fields. The Complementary Metal Oxide Semiconductor (CMOS) technology has contributed to high-speed computation and low power consumption, enabling AI to have a rich footprint in today's technology.

1.1 Artificial Intelligence

AI has changed how the world views technology. It can learn from the environment and act precisely to achieve its goals. More elaborately, we can define AI as the system's ability to understand external information, learn from the information, and use those studies to achieve required goals and tasks through malleable adaption. A classical AI mainly analyzes the environment and takes required actions that widen the chance of success. The goal of an AI is either simple or complex. AI has proved successful in achieving some impressive milestones like defeating the renowned world chess champion Garry Kasparov [1], winning different "Atari Games" or constructing SPAUN, an enormous 2.3 million artificial neuron model of the brain that mimics the behavior of a human [2]. AI has also played a massive part in solving some of the significant problems in the most knowledge-intensive areas like self-driving cars, adding intelligent speech-recognized personal assistants in smartphones like Apple's *Siri* [3], Google's *Google Assistant* [4], Microsoft's *Cortana* [5], Amazon's *Alexa* or even creating bots in YouTube to learn from the specific video generic.

AI development mainly revolves around the use of different algorithms [6]. AI consists of a cluster of simple algorithms that form a complex algorithm. Most of the AI algorithms can learn from the data using a cluster of algorithms, and the various branches of AI are shown in Fig. 1.1. The main sub-branch of AI is Machine Learning (ML), and the further sub-branch is DNN.

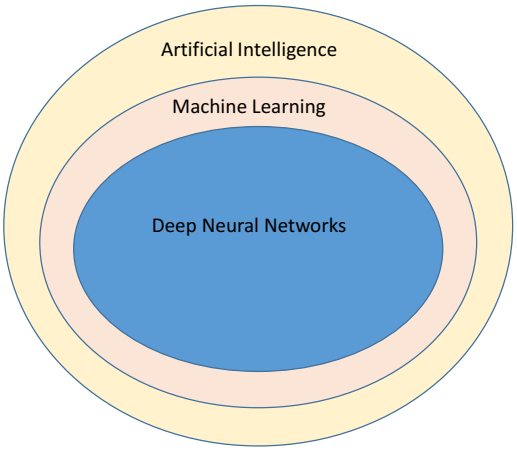


Figure 1.1: Various branches of AI.

1.2 Machine Learning

ML, a branch of artificial intelligence, learns from historical data to predict accurate new outputs. ML can improve any software application's output prediction without explicit coding. Instead, ML learns from a series of training algorithms to perform predictions on the input values and make accurate decisions without being programmed. There are a variety of applications used in ML, such as object recognition, object classification, and virtual personal assistants, where it is challenging to develop traditional algorithms to perform the required tasks.

The purpose of ML is for computers to learn how to conduct a particular task without being explicitly taught or programmed to do so. ML algorithms help computers learn to conduct specific tasks. Computers do not need to be expressly trained or specifically programmed to perform simpler operations like mathematical operations, running software, or playing music. However, it is challenging for humans to develop the necessary algorithms for more advanced tasks like self-driving cars or intelligent robots. Therefore, in practice, it is more effective to help the machine develop its algorithm than to have algorithms prescribed for every required step.

The branch of ML employs various approaches like supervised, unsupervised, and reinforcement learning to train computers to conduct specific tasks without a thoroughly trained algorithm. In most cases, a series of potential answers exist. It is necessary to label specific correct answers as accurate. This can be used as computer training data to improve the algorithm and determine accurate answers. The most common example is using the Modified National Institute of Standards and Technology Database (MNIST) dataset to learn from the series of handwritten digits to recognize digital characters. ML approaches are broadly classified into different categories, as shown in Fig. 1.2.

Supervised Learning: A computer provides examples of inputs and their corresponding desired outputs. The data, i.e., mainly offered, is the training data, which consists of a series of training examples. Each trained example has more than one input and the corresponding desired output, known as supervisory data. The training example is a feature vector, and the corresponding training data is a matrix. By constantly optimizing an objective function (the

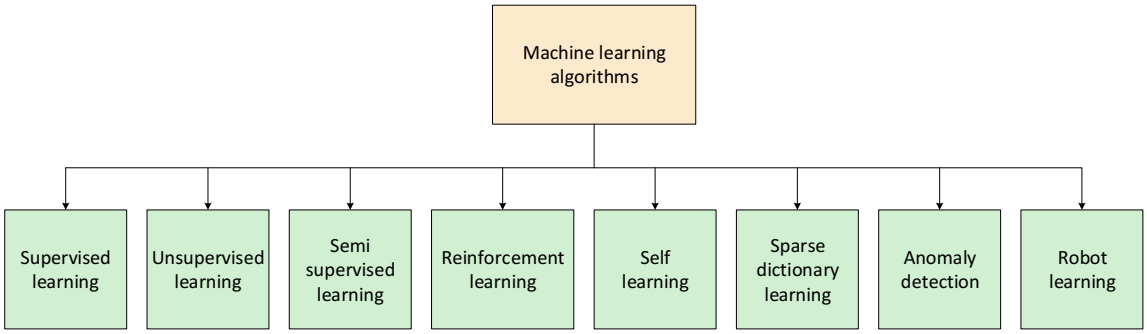


Figure 1.2: Various categories of machine learning.

objective function stands for some of the costs associated with an event or a value of one or more variables into an actual number), the supervised algorithms learn from a function that is valued to predict the desired output with a current set of inputs. The optimal function will allow the algorithm to determine the new outputs for the inputs that are not part of the corresponding trained data. Over time, the algorithm's prediction accuracy is expected to increase as it learns how to perform the task [7].

The supervised learning algorithms are active learning, classification, and regression. Classification algorithms are the most popular algorithms of this decade. They are employed when the outputs are constrained only to a defined set of values. Regression algorithms are used when the outputs have a specific numerical value under the desired range.

Similarly, active learning is between classification and regression. It aims to learn from the examples of a "similarity function" that decides how two objects are related. Visual identification, facial recognition, and speech recognition are among the applications of this technique.

Unsupervised Learning: No inputs or outputs are given to these learning algorithms. Instead, the algorithms must find their structures at their input. In unsupervised learning, the algorithm must discover the hidden patterns in the input data or near the end as feature learning. In unsupervised learning, the algorithms learn from test data that are not classified or labeled. In contrast to backpropagation and learning, unsupervised learning algorithms find certain commonalities in the input data and react to the output based on whether they exist in the new data. The main application for unsupervised learning is in the field of statistics, mainly in performing the probability density function [8].

Semi-supervised learning: Semi-supervised learning lies between supervised and unsupervised learning. A few examples of semi-supervised learning are missing training labels like image classification, text classification and speech analysis. However, many machine learning researchers have shown that labeled data, when used with unlabeled data, produces considerable learning accuracy [9].

Reinforcement Learning: Reinforcement learning is a branch of ML that identifies desirable behaviors and eliminates unwanted ones. Reinforcement learning recognizes and explains its environment, takes the necessary action, and learns through trial and error [10].

Self-Learning: The self-learning algorithm calculates in a crossbar manner where actions and emotions are dictated by their consequences. In this algorithm, cognition and emotion are

the ones that drive the entire system. Therefore, it mainly has one input, situation, action, and output [11].

Feature Learning: Feature learning algorithms try to store the information in the input and transform it internally to make it more useful. As a pre-processing step, this is done before performing classification or prediction. The feature learning technique can be supervised learning [7] or unsupervised learning [8]. A labeled input set is used to teach supervised learning. The best examples of supervised feature learning are Artificial Neural Networks (ANN) and Multi Layer Perceptron (MLP). Similarly, the features are learned from the unlabeled input data in unsupervised feature learning. Some examples of unsupervised feature learning are auto-encoders and dictionary learning [12].

Sparse Dictionary Learning: Sparse dictionary learning is another ML algorithm in the classification category used to decide to which class a previously imaginary training example belongs. An analogous dictionary is used to find the class of new training examples. The [12] uses this algorithm primarily for denoising images.

Anomaly detection: An algorithm is used for anomaly detection to detect unique items and events that change significantly from most of the actual data. Examples of anomaly detection include bank fraud and medical emergencies. There are three main classifications for anomaly detection. First, unsupervised anomaly detection detects the abnormality in the unlabeled set of data under the expectation that most of the instances in the data set are normal. The supervised anomaly detection technique requires a data set that has already been labeled as "normal" and "abnormal" and involves training a classifier. Finally, the semi-supervised anomaly detection technique requires constructing a model that represents the expected behavior from a given set of standard training data sets that tests the probability of a test example to be generated by the model [13].

Robot Learning: Robot learning algorithms generate learning experiences to aggregate new skills through self-exploration and social interaction with the environment [14].

1.3 Deep Neural Networks

DNNs are among the most important keystones of contemporary AI [15] and ML. DNNs are the heart of the machine learning system, as shown in Section 1.2. Due to modern DNNs rapid development, they can perform challenging tasks with unprecedented accuracies like image recognition [16] [17], speech recognition [18], keyword spotting [19], hand gesture recognition [20] and complex video games [21]. While the first DNNs were introduced in the 1960s with supervised, feed-forward MLPs, they became more popular after 2010. As of 2010, large-scale industrial applications have begun making a meaningful impact on the present-day world thanks to various training data, which led to the development of highly accurate specialized hardware. Furthermore, most Internet of Things (IoT) and communication devices focus on simplifying daily life with socially critical applications like auto sentence generators in Gmail, AI-driven photo editing in Instagram, and Google Maps for navigation. Consequently, extensive research is being conducted in industry and academia to make the applications smarter.

DNN revolutionizes the world by making it a better place and more accessible for day-to-day work. The powerful DNN algorithms can solve most of the complex tasks from the previous century. Therefore, the current area of focus is using DNNs to perform complex machine-learning functions that will help real-world applications. Nevertheless, this poses a significant challenge to the existing hardware systems and infrastructure. For example, existing general-purpose processors cannot perform complex estimations such as autonomous vehicles, innovative healthcare, robots, and smart assistants with the required performance and efficiency [22]. Standard processors are unable to handle the requirements of machine learning. Therefore, the AI chips have an added Neural Processing Unit (NPU) specifically designed to manage the machine learning data. As a result, AI processors offer high performance and better energy efficiency with diverse computing abilities. For example, image recognition and processing are faster when using smartphone NPUs, designed to perform multiple tasks efficiently. In addition, it can manage specific programming tasks more efficiently than general-purpose processors [22]. For example, in 2016, Apple Inc. CEO Tim Cook said, "AI will become even more of a personal assistant than it is today. So, where you are most likely not leaving your house without it today, you will be connected to it in the future." [23]. This statement points out the growing demand for hardware in DNN-related applications. In addition, dedicated hardware is imperative for meeting computational demands and reducing operational costs in today's fast-changing world. These high demands have aroused the efficient development of dedicated DNN accelerators that meet accuracy, computational, and energy-efficient requirements[24].

In contrast with other standard technologies like cyber security, video coding, wireless communication, and blockchains, DNNs are growing faster. A high number of scientific papers have been published on new DNN models that perform more complex tasks with much higher accuracy than the present State of the Art (SoA). Different DNN models offer a distinct set of challenges:

Memory: A typical DNN has a memory of hundreds of MBs to GBs. Hence, storing these models requires high memory resources, and the transmission cost involved in transmitting the information through a communication channel with a limited memory capacity is incredibly challenging.

Execution Speed: Due to layers and increased data movement, some DNN models for speech and image recognition and natural language processing have high latency. Such latency will impact the positive user experience in the entertainment field and have a critical implication in healthcare and autonomous driving. For example, GoogleNet [25] for image classification takes around 0.006 seconds to classify one image on the Intel CascadeLake.

Energy: Most of the DNN models that perform complex tasks have many parameters for better accuracy. This will lead to increased data transfer between off-chip and on-chip memory elements, increasing the power consumption of the hardware [26]. In most cases, the data movement has 2× the energy cost of the computation cost. For example, as explained in EIE [27], the 32-bit arithmetic operation consumed a total of 12.1pJ of energy, whereas the 32-bit Dynamic Random Access Memory (DRAM) data movement consumed a total of 640pJ. As a result, the models with low energy efficiency will not fit on resource-constrained devices like mobile phones, VR glasses, and wearable smartwatches.

Cost: To compute the DNN models according to the above criteria will require an expensive and large hardware device.

The hardware, in terms of bandwidth and speed inferencing a powerful DNN model, should be capable of handling memory requirements for ample storage of data, efficient computation techniques with lower latency, and improved data transfer to reduce the energy and computation cost. In addition, each DNN model has its configurations and sizes. Thus, building dedicated hardware for one DNN model could be more efficient. So, in addition to accuracy, energy efficiency, and high performance, the flexibility of inferencing different DNN models is also an especially crucial factor.

1.4 Need for efficient processing of DNN models

As mentioned in Section 1.3, even though DNNs are capable of previously defined delivering SoA accuracy on the majority of AI tasks, they come at a very high computational complexity. As a result, to infer the power of DNN models, there is a need to enable the efficient processing of DNN to improve energy efficiency and throughput without compromising accuracy. Among other things, processing efficiently requires choosing between spatial and temporal architecture, minimizing data movement between off-chip DRAM and on-chip Static Random Access Memory (SRAM), improving computation techniques, especially for MAC operations, and improving implementation of different data flows.

1.5 Scientific Contributions

This thesis stresses the development of energy-efficient hardware architectures for deep learning models and compressed deep learning models. Each contribution to the thesis can be summarized as follows.

- In Chapter 3, we provide a detailed analysis of different data flows used to infer the DNN models efficiently. Each data flow has a wide variety of trade-offs between throughput, energy efficiency, and performance. This chapter provides detailed implementation techniques for each set of data flows, including the advantages and disadvantages of inefficient hardware design. Also, in this chapter, we explain energy-efficient ways of data handling for hardware implementation to support different layers of neural networks like CONV and Fully Connected (FC) layers. We finally conclude with the different experimental results for different data flows and show which data flow would be used for the efficient implementation of the hardware design [28] [29] [30] [31] [32].
- In Chapter 4, we explain the power-efficient multi-bit accelerator for memory-prohibitive DNNs. In this chapter, we explain the multi-bit accelerator that reduces the energy and power requirement by employing the truncation of the Partial Sums (PSums) of the preceding layer before feeding it into the next layer. The multi-bit accelerator starts by inferencing the 32 bits for the first convolution layers and sequentially truncating the bits on the MSB/LSB of integer and fractional parts without further training on the original network. At the last FC layer, the top-1 accuracy is maintained with the reduced bit width of 14 and top-5 accuracy up to 10-bit width [30] [31].

- In Chapter 5, a compression technique is proposed to reduce the silicon area and power requirements of MLPs with high predictive performance. Firstly, a novel hardware architecture named FantastIC4 is discussed, which (1) supports the efficient on-chip execution of multiple compact representations of FC layers and (2) minimizes the required number of multipliers for inference down to only four. Moreover, to make the models amenable to efficient execution on FantastIC4, a novel constrained training method is introduced that makes for efficient 4-bit quantization and a highly compressed DNN model. The software development for the entropy constraints and compression of the DNN algorithms was designed by the authors Mr. Simon Wiedemann and Mr. Daniel Becking, and the entire hardware development to support these algorithms was done by Mr. Suhas Shivapakash, who is the main author of the thesis. More information about the thesis contribution is explained in the below section.

1.6 Thesis Contributions

This thesis's primary focus was creating efficient hardware architectures for the DNN models. That concentrated on efficient data movement by improving the performance of Terra Operations Per Second (TOPS) and reducing the total power consumption. The architectures were implemented in the Field Programmable Gate Arrays (FPGA) and the Application Specific Integrated Circuits (ASIC).

The thesis contributions for the chapters are condensed as follows:

- For chapter 4:
  - **Mr. Shivapakash's contribution:** He researched, developed, and designed the data flow architecture for the multi-bit accelerator, the truncation techniques, FPGA implementation, and the complete physical design involved in the ASIC implementation.
  - **Co-author's contribution:** Mr. Hardik Jain provided the trained data set for the different DNN models like AlexNet, EfficientNet, SqueezeNet, and MobileNets. All the co-authors, Hardik Jain, Prof. Olaf Hellwich, and Prof. Friedel Gerfers, provided critical feedback that helped in the development of the conference and journal manuscripts [30], [31] on which the multi-bit accelerator chapter is based.
- For Chapter 5:
  - **Mr. Shivapakash's contribution:** He developed the hardware architecture and the data flow movement to infer the CSR and entropy-constrained DNN algorithms for the FantastIC4. Moreover, he was also involved in the hardware algorithm development for the Accumulate-Multiply (ACM) computational flow and floating-to-fixed point conversions. In addition, he implemented both the FPGA implementation and the ASIC implementations for the FantastIC4.
  - **Co-author's contribution:** Mr. Simon Wiedemann and Mr. Daniel Becking's main contribution was engaged in the software development of the CSR to Bit Mask Conversion, Entropy constrained algorithms, and Decoder techniques. Mr. Simon

Wiedemann and Mr. Daniel Becking were also involved in the ablation studies for the FantastIC4 hardware implementation. Along with them, Mr. Pablo Wiedemann, Prof. Samek Wojciech, Prof. Friedel Gerfers, and Prof. Thomas Wiegand provided valuable feedback that helped in the development of the journal publication [32] on which the FantastIC4 chapter is based upon.

1.7 Thesis Organisation

The thesis is organized as follows:

- **Chapter 2:** In Chapter 2 the theoretical background of the DNNs including its architecture, training, different operations, different layers and the mathematical background of the various layers is provided. Moreover, we provide extensive information on the different DNN development resources and prominent datasets.
- **Chapter 3:** In Chapter 3, a detailed analysis of different data flows involved in the efficient processing of DNNs is explained. In addition, we provide a detailed implementation technique for other data flows. Furthermore, we will explain the different data handling methods for energy-efficient hardware design to support different layers of neural networks. Finally, we end the chapter with the required metrics for DNN training and inference.
- **Chapter 4:** In Chapter 4, we explain the power-efficient multi-bit architecture for inferencing the memory-prohibitive deep neural networks. This chapter describes the multi-bit architecture that truncates the PSums results of the previous layer before it is fed as an input to the next layer. Here, we provide the efficient implementation of the PE array arranged in a systolic array, where the MAC implementation is performed through approximate computing. The experimental section shows how to truncate bits between the integer and the fractional part with low RMS error so that the top-1 and top-5 accuracy is unaffected. Finally, we provide the FPGA and ASIC implementation results and compare our architecture with the present SoA. A majority of the contributions of this chapter are based on published journal articles, where the author of this thesis is the first author of the following publications [32].
- **Chapter 5:** This chapter introduces the FantastIC4, which can utilize low entropy statistics to enhance the inference accuracy of a compressed model. This chapter explains a detailed architecture for managing the SoA compressed models targeting the 4-bit quantization and Huffman encoding. The experimental results of this chapter explain the different benchmarking tests and compare our FantastIC4 architecture with other SoA architectures. Finally, we provide the FPGA and ASIC results of the FantastIC4 and show the effect of entropy on the dynamic power consumption. A majority of the contributions of this chapter are based on published journal articles, where the author of this thesis is one of the main authors of the following publications [32], where the software development for the algorithms explained in this chapter was developed by Mr. Simon Wiedemann and Mr. Daniel Becking and the hardware development to support these algorithms was done by Mr. Suhas Shivapakash.

- **Chapter 6:** Finally, the last chapter summarizes the obtained results and conclusions of the thesis and provides the direction for future work in the energy-efficient hardware architecture for deep learning.

2

Deep Neural Networks

2.1 Introduction to Deep Neural Networks

Deep learning belongs to a larger class of machine learning algorithms. These algorithms can engage in supervised [7], semi-supervised [9], and unsupervised [8] learning techniques, as explained in Chapter 1. Different architectures exist for deep learning, including DNNs, Recurrent Neural Networks (RNNs), and Deep Belief Networks (DBNs). Computer vision, image recognition, speech recognition, natural language processing, social network filtering, language translation, bioinformatics, radar, and military surveillance use these architectures [33] [17]. A DNN mimics how information is processed and communicated in the human brain [34]. Information processing and communication imply how the brain processes different aspects of information, such as image color, image contrast, and distinguishing between other objects. Communication refers to sending electrical signals from the brain to distinct body parts in response to changes in the surrounding environment. The adjective "deep" in a deep neural network comes from the deep usage of multiple layers in the neural network.

DNNs offer a unique set of advantages compared to the traditional computer vision algorithms like Hough transforms and Geometric hashing that were not evident before the turn of this decade [35]. Instead of the arduous or the hit-and-miss approach of creating a near distinct algorithm capable of solving each problem in a particular domain, we can train these algorithms to learn from the raw data by a process called *training* to handle a new set of problems. Within these DNN fields, it is often called brain-inspired computation, as these algorithms learn to solve the problems independently. The DNNs, brain-inspired computations, take certain aspects of the brain's functionality and emulate its function in solving newer problems.

Scientists across the world are still researching the functionality of the brain. However, the main computational element of the brain is the "neuron". There are around 100 billion neurons inside the brain. These neurons are connected by several elements called axons and dendrites. Dendrites are the elements that enter the brain, and axons are the elements that

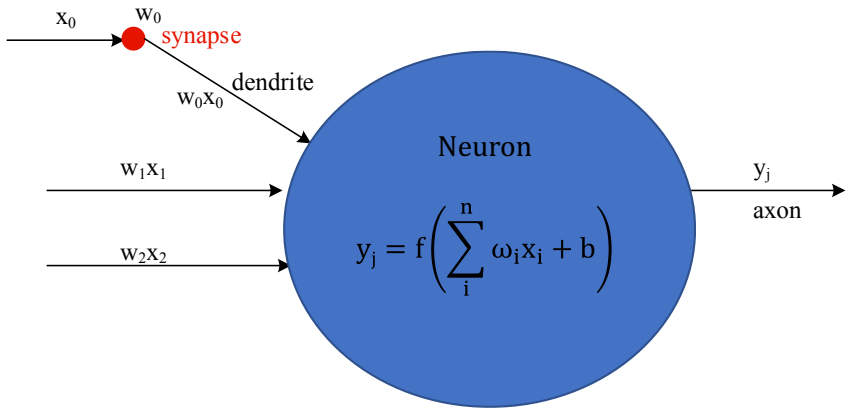


Figure 2.1: Connections to a neuron in the brain. x_i, w_i, f(MAC functionality), n and b are the activations, weights, non-linear function, size of the activation matrix, and bias respectively [29].

leave the brain. A neuron accepts the signals via dendrites, performs the computation, and generates the output signals called axons. The incoming and outgoing signals are called activations. The axon is the output of one neuron while it is connected to the dendrites of the other neurons. The connection between a branch of the axon and a dendrite is called the synapse, as shown in Fig. 2.1. Around 10^{14} - 10^{15} synapses exist in the average human brain. The main characteristic of the synapse is that it can extend the number of input signals x_i crossing it as shown in Fig. 2.1. The extension factor is referred to as the weight w_i, and in this way, the brain is continuously learning from the changes in the weights associated with the synapses [29]. The different weight values will result in a different response to the input activation. As shown in Fig. 2.1, the range of "i" ranges from 0 to the maximum size of the neural network layer under the inference. For example, in the AlexNet for the first convolutional layer, the maximum length of "i" is 225 × 225.

The computation of the weighted sum of the input activation values influences most current neural networks. Each weighted sum correlates to the value extension performed by the synapse and connects those sets of values to a neuron. Each neuron does not just output the weighted sum but combines the non-linear function with the weighted sum to generate the final output.

The neural networks use multiple layers to extract higher-level features from raw inputs. For example, in image classification networks, the lower layers of the neural network identify the edges, and the higher layers identify the relevant concepts like faces, digits, and letters. These networks mainly consist of neurons, synapses, weights, biases, and functions. DNNs are trained to identify the class of particular objects and calculate the probability of that class in the object. Then, the end-user can review the results, select the probability the network should display, and return the proposed set of classes. Each mathematical manipulation in the neural network is considered a label, and complex DNNs have many hidden layers inside them, hence the name "Deep Neural Networks". An iteration of a simple neural network with three hidden layers is shown in Fig. 2.2.

In most cases, DNNs have more than a thousand hidden layers embedded inside the network for an efficient learning process. Each layer in the DNNs has a weighted sum of inputs, termed

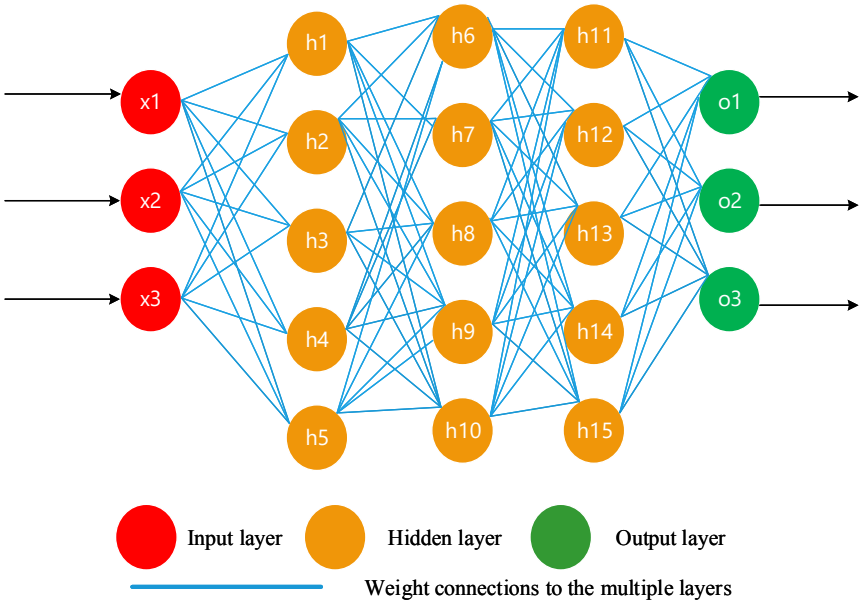


Figure 2.2: Layers of DNN.

activations, as shown in the above Fig. 2.1. These activations and weights perform a defined set of mathematical operations called the MAC, followed by non-linear functions to generate the final output activations as explained in Fig. 2.1. The different non-linear functions that are used in DNN training and inference are:

- **Sigmoid:** Sigmoid functions are asymmetric, differential, and real functions with no negative derivative at each point and the inflection point for all input values [36]. The sigmoid function is defined in Eq. 2.1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

  This approach has the main benefit of making the function more monotonic and differentiable. ML models are widely used to predict output values as a probability when the values are between zero and one. The disadvantage of the sigmoid function is that the outputs are not zero-centered and are computationally costly, especially on the hardware.

- **Hyperbolic tangent:** As with sigmoid functions, hyperbolic tangent functions are defined between +1 and -1 but are zero-centered [37]. The hyperbolic tangent function is defined in Eq. 2.2.

$$f(x) = \frac{e^{x} + e^{-x}}{e^{x} - e^{-x}} \tag{2.2}$$

  Its main advantage is that the derivative is steeper and has more values to compute. Therefore, ML models can be computed faster. However, the main disadvantage is that it is computationally expensive.

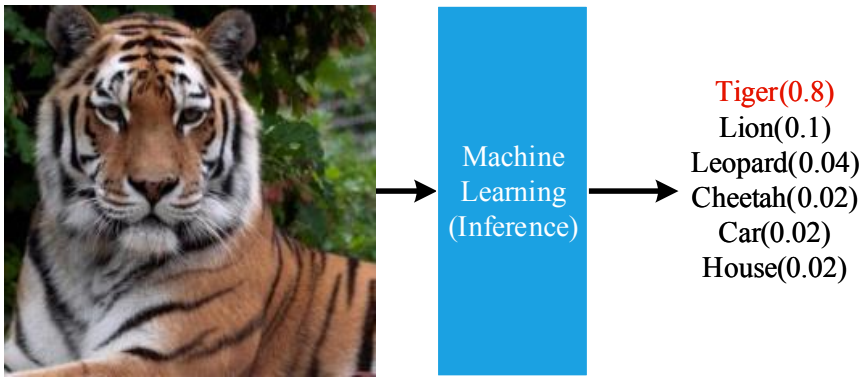


Figure 2.3: Image classification example. The machine learning algorithm takes the input of the image and outputs the probability scores for a defined set of previously trained classes. The source of the Tiger picture is ©Google.

- **Rectified Linear Unit (ReLU):** ReLU technique is computationally cheaper and hardware friendly [38]. It is defined with a simple function as shown in Eq. 2.3.

$$f(x) = max(0, x) \tag{2.3}$$

  However, one slight disadvantage is that the function's gradient becomes zero when the inputs approach zero or negative. This will prevent the ML models from performing the backpropagation technique and learning from the error.

The diagrams for each non-linear function are shown in Fig. 2.13, according to Fig. 2.1, In activations(x), "i" varies from 0 to the maximum size (N) of the activation matrix, such as 227 × 227 for AlexNet; similarly, in weights(w), "i" varies from 0 to the maximum size (N) of the weight matrix, such as 13 × 13 for AlexNet. So, let us consider two simple examples to depict the functionality of Fig. 2.1, where i ranges from 0 to 1.

Example 1:

Let $w_0 = 10, w_1 = 11, x_0 = 5, x_1 = 3, b = 20$ So, the output y is

$y = f(10 \times 5 + 11 \times 3 + 20)$

$y = f(103)$

Here f(x) is a ReLU function as shown in Eq. 2.3

So, the output y is 103.

Example 2:

$Let w_0 = 1, w_1 = 2, x_0 = 3, x_1 = 1, b = -6$. So, the output y is

$y = f(1 \times 3 + 2 \times 1 - 6)$

$y = f(-1)$

Here, the value of x is -1, so the output y is 0.

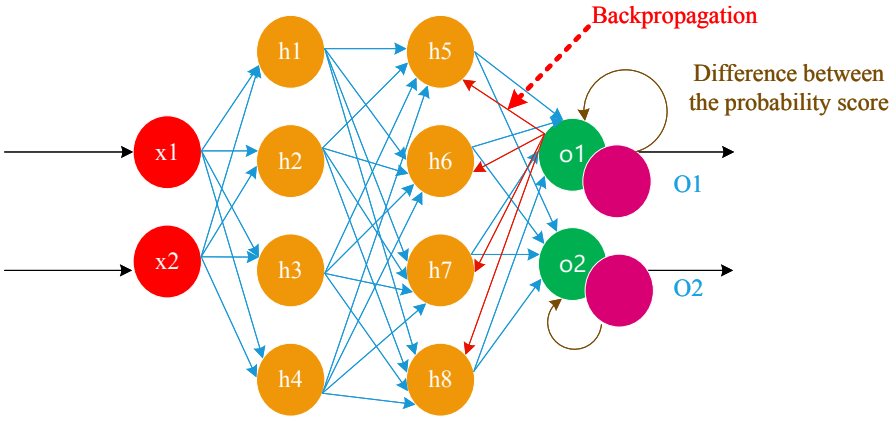


Figure 2.4: Backpropagation example.

2.2 Inference versus Training of the DNNs

The DNNs are part of the machine learning algorithm, as explained in Chapter 1. The basic functionality stays the same as it learns from the given set of tasks to perform a new task. In most cases, learning about the DNNs involves determining the value of the network's weights, activations, and biases. The process is known as network training. After the training, a hardware/program can compute the network's output by using the activations, weights, and biases generated during the training, running a hardware/program with the generated weights, activations, and biases.

An image classification example in Fig. 2.3 is the best example for DNN inference. When we perform the inference of the DNNs, we provide an input image, and the output from the DNN will be a probability of scores for each object class. The class with the highest probability will show the most likely class of the object in the image. The main goal of training is to find the weights that would maximize the probability score of the correct object class and minimize the score of the incorrect object class. The correct object class is always known during the network training because it is provided for the images used. The difference between the actual probability scores and those computed by DNN is mainly based on the current weights referred to as loss (L), as shown in Eq. 2.4. Hence, the main goal of training is to find the proper set of weights that would reduce the average loss over a complete training set.

The weights are usually amended during the network training using a hill-climbing optimization technique called gradient descent. The relative loss for the multiple gradients of each weight is the partial derivative of the loss concerning the weight used to update the new weight, as shown in Eq. 2.4. Here α is called the learning rate. The gradient indicates how the loss can be reduced by changing the weights. The process repeats iteratively to minimize the overall loss [29]. A competitive technique to evaluate the partial derivatives or the gradient is called *backpropagation*.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial L}{\partial w_{ij}} \tag{2.4}$$

2.2.1 Backpropagation

Backpropagation is a chain of partial derivative techniques used to fine-tune the weights of a neural network based on the error rate obtained from the previous iteration. The error rates will be reduced by fine-tuning the weights, and the neural network will be more robust during the general classification. The backpropagation example is shown in Fig. 2.4. The input activations X1 and X2 arrive in a pre-connected path, and the activations are modeled with weights according to Fig. 2.1. The final output activations are generated after passing through the hidden layers. The difference error (θ_B) is evaluated according to Eq. 2.5 and θ_B is backpropagated to fine-tune the network until the error is minimized.

$$\theta_B = \theta_{actual} - \theta_{desired} \tag{2.5}$$

The backpropagation computation is a simple technique that is fast and easy to program. Since the training dataset sums to zero, backpropagation tends to learn faster. The mean value of each input variable can be subtracted from it to achieve this. The confluence of iteration usually runs faster when the average of each input variable over the training set is close to zero. The backpropagation technique is used to backpropagate through each of the hidden layers:

- Compute the gradient loss compared to the weights from the input activations and the loss gradients compared to the output activations.
- Finally, compute the gradient loss relative to the input activations from the weights and the gradient loss compared to the output activations.
- Repeat the process until the error is minimal.

The backpropagation technique inherently requires preserving the network's transitional outputs for backward computation, which increases the overall storage requirements. Also, since the gradients use a hill-climbing approach, the precision needed for training will always be higher than that needed for inference.

Apart from backpropagation, other techniques like Elman Neural Network [39] and Jordan Neural Network [40] are used to improve the robustness and efficiency of the training. However, they are less popular than the backpropagation technique. The loss from the multiple input data sets is collected before updating the weights. This speeds up and stabilizes the complete training process.

Let us consider a simple example of the backpropagation algorithm shown in Fig. 2.5. Firstly, let us calculate the total output value for the net h_1:

$net_{h_1} = w_1 * x_1 + w_2 * x_2 + b_1 * 1$

$net_{h_1} = 0.14 * 0.07 + 0.26 * 0.11 + 0.36 * 1$

$net_{h_1} = 0.3984$

Applying non-linear function, the ReLU according to the Eq. 2.1, the out_{h_1} will be

$out_{h_1} = \frac{1}{1+e^{-net_{h_1}}} = \frac{1}{1+e^{-0.3984}} = 0.598303$

Similarly, the out_{h_2} will be:

$net_{h_2} = w_3 * x_1 + w_4 * x_2 + b_1 * 1$

$net_{h_2} = 0.37 * 0.07 + 0.42 * 0.11 + 0.36 * 1$

$net_{h_2} = 0.4321$

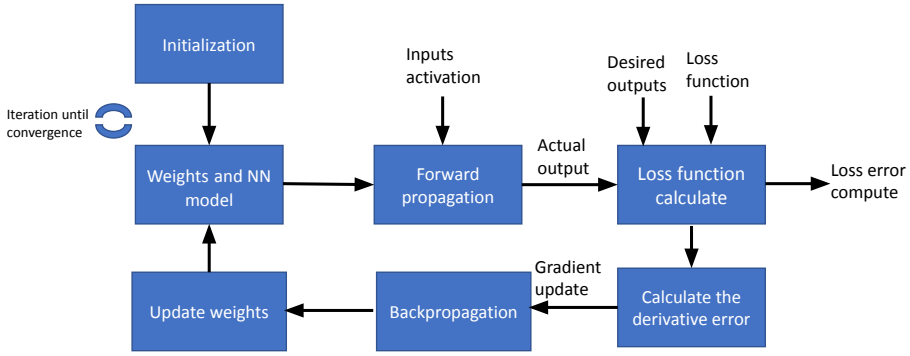


Figure 2.5: iteration for backpropagation algorithm.

$out_{h_2} = \frac{1}{1+e^{-net_{h_2}}} = \frac{1}{1+e^{-0.4321}} = 0.606375$

After computing the hidden layers, we can now calculate the output layers out_{o_1} and out_{o_2}.

$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$

$net_{o_1} = 0.18 * 0.598303 + 0.24 * 0.606375 + 0.6 * 1$

$net_{o_1} = 0.85322454$

$out_{o_1} = \frac{1}{1+e^{-net_{o_1}}} = \frac{1}{1+e^{-0.85322454}} = 0.701243$

$net_{o_2} = w_7 * out_{h_1} + w_8 * out_{h_2} + b_2 * 1$

$net_{o_2} = 0.36 * 0.598303 + 0.62 * 0.606375 + 0.6 * 1$

$net_{o_2} = 1.19134158$

$out_{o_2} = \frac{1}{1+e^{-net_{o_1}}} = \frac{1}{1+e^{-1.19134158}} = 0.766980$

Now, we will calculate the total error:

$Error_{total} = \sum \frac{1}{2} (target - actual)^2$

The target outputs for o_1 is 0.02, o_2 is 0.98 and the error for the net output o_1 and o_2 are:

$Error_{o_1} = \sum \frac{1}{2} (0.02 - 0.775416)^2 = 0.232204$

$Error_{o_2} = \sum \frac{1}{2} (0.98 - 0.766980)^2 = 0.0022688$

The total error will be:

$Error_{total} = Error_{o_1} + Error_{o_2}$

$Error_{total} = 0.232204 + 0.0022688 = 0.2344728$

Now, we employ the backpropagation algorithm to update each weight of the DNN model so that the actual output is equal to the target output, thereby reducing the total errors in the neuron model's output. The output o_1 is dependent weight w_5, so we need to decide how much change in w_5 affects the total error of o_1. The total error for the weight w_5 is calculated according to the chain rule as shown below:

$\frac{\partial Error_{total}}{\partial w_5} = \frac{\partial Error_{total}}{\partial out_{O_1}} * \frac{\partial out_{O_1}}{\partial net_{O_1}} * \frac{\partial net_{O_1}}{\partial w_5}$

We know that,

$Error_{total} = \sum \frac{1}{2} (target_{o_1} - out_{o_1})^2 + \sum \frac{1}{2} (target_{o_2} - out_{o_2})^2$

Therefore,

$\frac{\partial Error_{total}}{\partial out_{o_1}} = 2 * \frac{1}{2} (target_{o1} - out_{o1})^{2-1} * -1 + 0$

$\frac{\partial Error_{total}}{\partial out_{o_1}} = -(target_{o1} - out_{o1}) = -(0.02 - 0.701243) = 0.681243$

Similarly the partial derivative of o_1 with respect to net_{o_1} would be:

$out_{h_1} = \frac{1}{1+e^{-net_{h_1}}}$

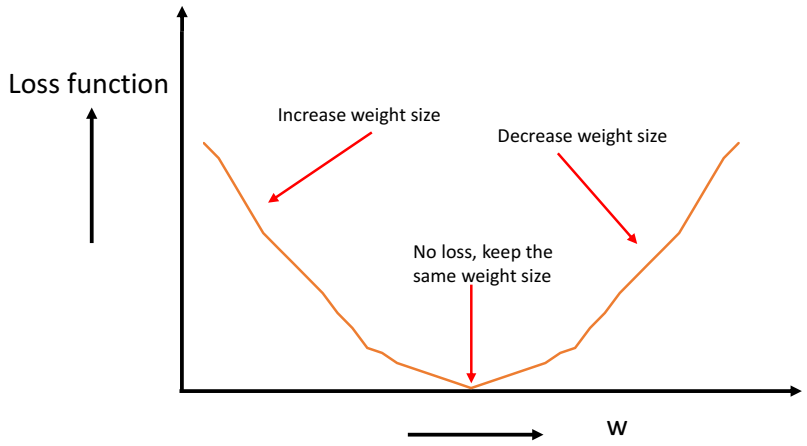


Figure 2.6: Loss function vs Weight update.

$\frac{\partial out_{o_1}}{\partial net_{o_1}} = out_{o_1}\,(1 - out_{o_1}) = 0.701243(1 - 0.701243) = 0.209501255$
Finally, now we can determine how much net o_1 changes concerning the weight w_5
$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$
$\frac{\partial net_{o_1}}{\partial w_5} = 1 * out_{h_1} * w_5^{(1-1)} = out_{h_1} = 0.598303$
Now grouping them,
$\frac{\partial Error_{total}}{\partial w_5} = \frac{\partial Error_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$
$\frac{\partial Error_{total}}{\partial w_5} = 0.681243 * 0.209501255 * 0.598303 = 0.08539056$
To decrease the error, we can refer to the Eq. 2.4, hence the updated weight w_5^+ would be:
$w_5^+ = w_5 - \alpha \frac{\partial Error_{total}}{\partial w_5} = 0.18 - 0.7 * 0.08539056 = 0.12022$
Similarly,
$w_6^+ = 0.185980006$
$w_7^+ = 0.375945074$
$w_8^+ = 0,636160197$
Next, we will update the weights w_1, w_2, w_3 and w_4. So, update the weight w_1 will be,
$w_1^+ = 0.13654519$
Similarly,
$w_2^+ = 0.25467678$
$w_3^+ = 0.375945074$
$w_4^+ = 0.41449875$

After updating the weights, the final output o_1 is 0.776454 o_2 is 0.761276, the final If we repeat the process with the forward pass and the backpropagation more than 10000 times, we will come near to the actual target output. To depict this iteration, Fig. 2.6 shows how the loss function varies with the weight update, and Fig. 2.5 shows the block diagram of the entire backpropagation algorithm and overall weight calculation procedure.

2.3 History of the Deep Neural Networks

The foundation stone for the DNNs was laid in the early 1940s by Walter Pitts and Warren McCulloch when they formulated a mathematical model of a biological neuron in their paper "A Logical Calculus of the Ideas Immanent in Nervous Activity [41]." In 1957, Frank Rosenblatt

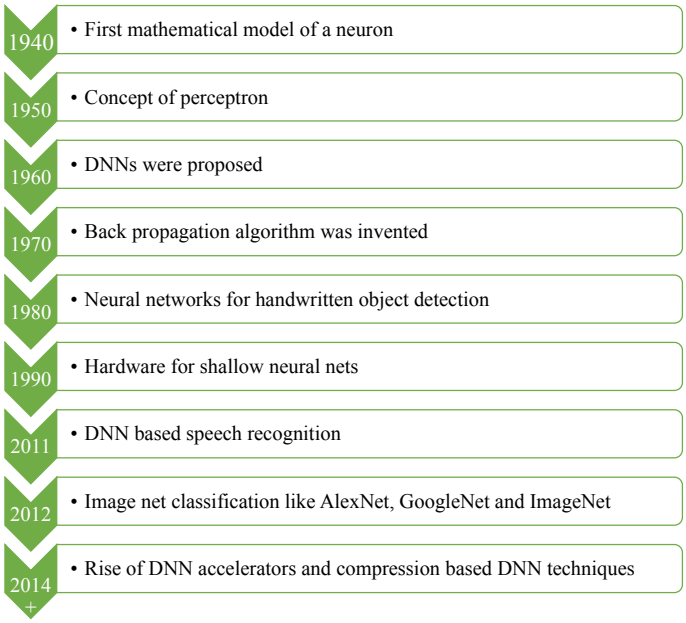


Figure 2.7: A terse history of neural networks.

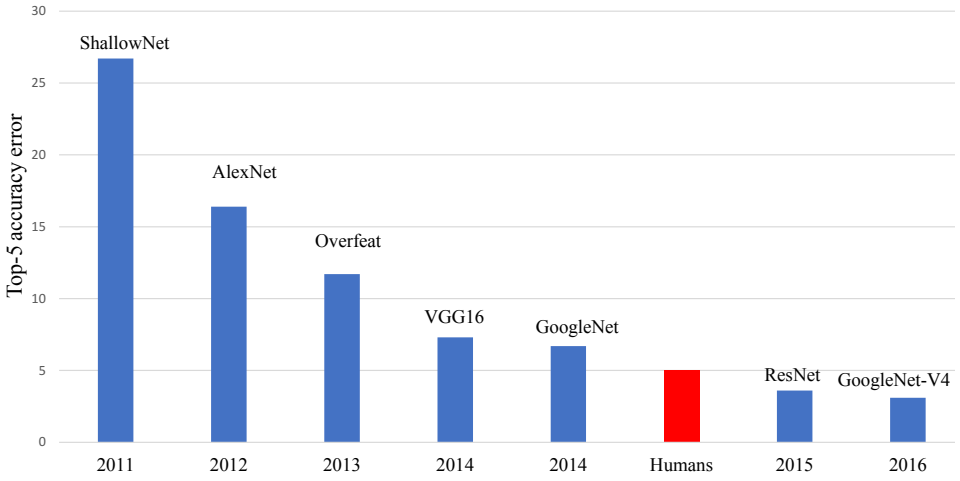


Figure 2.8: ImageNet challenge. The y-axis on the top-5 accuracy indicates the total % error.

established the concept of perceptron. Around the 1960s and 1970s, the idea of backpropagation and deep learning was invented. The first major practical applications of the DNNs were started in the late 1980s with the invention of LeNet for handwritten digit recognition [42]. The breakthrough in the technology using DNNs began around the 2010s with significant highlights such as Microsoft's speech recognition system and AlexNet image classification system. A concise history of neural networks is shown in Fig. 2.7.

The success of deep learning in the early 2010s is mainly due to three factors. The first factor is the availability of a large amount of information to train a neural network. Much training data is needed to learn a new powerful representation. Currently, Google Photos and Facebook have access to more than 500 million images per day are required. In addition, Amazon has access to 10 Petabytes of customer shopping information, and YouTube has more

than 1000 hours of videos uploaded every minute. Hence, most cloud service providers have access to many data to train their algorithms.

The second factor is the available computation capacity, which is the most critical factor. The advancement in CMOS technology and path-breaking research in computer architecture have led to increased computing ability. This has reduced the inference time of the powerful DNN models and helped fine-tune the network's training process to obtain improved results.

The major success in the early DNN applications has opened a new pathway for algorithm development. In addition, the open-source frameworks have helped many researchers explore the use of DNNs and make them much more efficient in solving complex tasks. Finally, combining all these efforts has led to new algorithmic techniques. This third factor has improved the application accuracy and broadened the range of applications for DNNs.

Another example of the advancement in deep learning is the ImageNet challenge. This challenge evaluates different algorithms for object detection and image classification on a large scale. In the image classification task, the algorithms are given an image, and they must classify what an image is, as shown in Fig. 2.3. The entire training set consists of around 1.2 million images, each labeled with one of the thousand object classes that the image contains. Then, in the evaluation phase, the algorithm must correctly identify the class of objects in a new set of images.

The most accurate networks that won the ImageNet contest over the years are shown in Fig. 2.8. From the graph, it becomes evident that the accuracy of the algorithms had an error rate of more than 25%

AlexNet utilized a powerful Graphics Processing Unit (GPU) to evaluate their algorithm, reducing the error rates. In 2012, only four entrants used GPUs to fine-tune their algorithm. However, by 2014, the use of GPUs became so dominant that most entrants (around 110) used them. This trend gradually shifted from the traditional computer vision approach to a deep learning-based one.

In 2015, ResNet [16] eclipsed human accuracy with a top-5 error rate below 5%. Today, the error rate is below 3 percent, focusing on 3-D image classification and object detection. These achievements are a primary contributing factor to the extensive range of applications that employ DNNs.

2.4 Applications of DNNs

Many applications, ranging from multimedia to the military, are using DNNs. In this section, we provide a brief overview of the areas in which DNNs are currently making an impact and highlight the areas where DNNs can potentially impact the future.

2.4.1 Image and video applications

Video is one of the most significant forms of big data. It accounts for over 70%-80% of today's internet traffic. For example, almost over 1000 million hours of video are collected around the world for video surveillance [43]. Computer vision techniques are essential to extracting information from videos. DNNs have helped improve the accuracy of computer

vision tasks such as image classification, object detection, action recognition [44], and image segmentation [45].

2.4.2 Speech and natural language processing

DNNs have significantly improved machine audio translation accuracy and related tasks like speech recognition, natural language processing, and audio generation. The efficiency of the MLPs has dramatically boosted the performance of the Keyword Spotting (KWS) applications [46].

2.4.3 Robotics

DNNs have successfully performed robotic tasks and motion planning for ground robots, visual recognition during the calamity, stabilizing quadcopters' control, and autonomous driving strategies [47].

2.4.4 Gaming

The use of DNNs has overcome challenges in the field of gaming. Innovations were performed in the training techniques for gaming applications, mainly on reinforcement learning. DNNs have beaten human accuracy in playing games like Alpha Go and Atari. By playing these games, DNNs have proven that the technology can perform an exhaustive search of different possibilities needed to make a different number of moves [48].

2.4.5 Medical

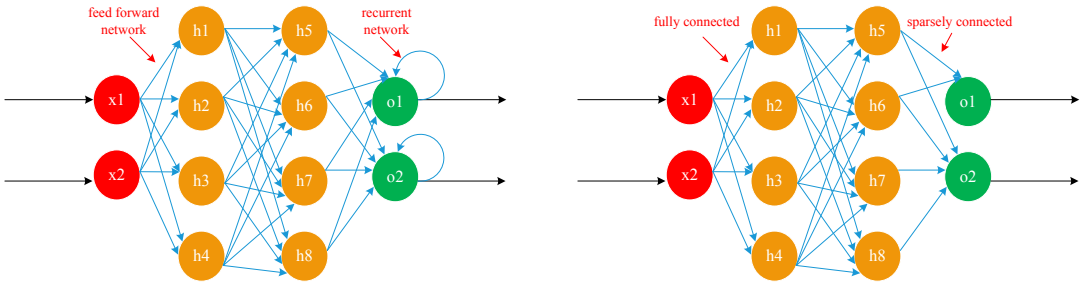
DNNs have played a vital role, for instance, in detecting cancer at an early stage, which has saved many lives. They have also helped in lesion detection and organ segmentation [49].

2.4.6 Other Applications

DNNs are used extensively in many multimedia applications. In the future, DNNs will play a more significant role in the medical and robotics fields, finance, infrastructure, weather forecasting, and event detection. The different application areas will provide new challenges for the efficient processing of DNNs, both in training and inference. With the novel challenges, there will be new DNN techniques that will be adaptive and scalable to meet the new requirements [50].

2.5 Embedded and cloud applications for DNN computation

For DNN processing, both inference and training have different computational requirements. Training requires many relevant datasets and essential computational resource requirements to estimate the multiple weight generation and bias values. In most cases, training a DNN model takes from several hours to numerous days; hence the training is typically performed on the cloud [27] [29] [51] [52] [53] [54]. However, inference can happen on cloud or edge applications like IoT or mobile phones.



(a) Feedforward v/s Recurrent (b) Fully connected v/s Sparsely connected

Figure 2.9: Different types of neural networks.

In most applications, placing the DNN inference processing near the sensor is proper. For example, in computer vision applications like predicting traffic patterns or understanding the user shopping experience in an e-commerce platform, it is more meaningful to extract the information from the video at the input of the sensor rather than the cloud to minimize the communication cost [29]. On the other hand, cloud processing is not desirable in time-critical applications like autonomous driving, robotics, and drone navigation due to the latency and security risks involved. Moreover, even though the processing speed is more related to training in these applications, it also affects inference since the performance [29] should be faster and more reliable to avoid potential hazards. However, videos involving a large amount of data computationally make processing overly complex. Hence, performing hardware is critical for the processing of these data.

Speech recognition applications help us connect with electronic devices such as smartphones conveniently. Most of the processing of these applications happens in the cloud, for example, Siri from Apple and Alexa from Amazon. However, to reduce latency and dependency on connectivity and improve security and privacy, performing the recognition on the smart device is appropriate.

Most embedded devices that perform DNN inferencing have memory cost limitations, stringent energy requirements, and area constraints. So, it is imperative to process the DNNs efficiently. Hence, this thesis will focus on the energy-efficient inference of the DNNs.

2.6 Overview of DNNs

Depending on the application, DNNs come in various shapes and sizes, which are changing rapidly to improve accuracy and efficiency. Most of the time, the input to the DNN is a collection of values defining the information to be analyzed. The input values can be a pixel of an image, sampled amplitudes of an audio signal, or the numerical values of the system.

The network that processes the input will come in two primary forms, a feed-forward network, and a recurrent network, as shown in Fig. 2.9(a). In feed-forward networks, most of the computation is performed as a sequence of operations on the previous layer's outputs. The final set of operations on the last layer will generate the network's final output. For example, the two instances are the probability that an image has a particular object and the probability that the audio signal contains a specific word like "Hey Siri." These feed-forward networks

don't include any memory. Thus, the number of outputs and inputs stays the same, regardless of the previous sequence of inputs.

In contradiction, recurrent neural networks (RNNs), among which Long Short-Term Memory networks (LSTMs) have internal memory to allow the long-term dependencies to affect the final output. In RNNs, the intermediate operation results are internally stored inside the network and used to input the next operations layer. However, this thesis focuses on the feed-forward networks since most of the computation in RNNs is the weighted sums, which is the MAC computation covered in the feed-forward networks. Also, there needs to be more attention and importance on the hardware acceleration in the RNNs [55] [56].

DNNs are also composed of only FC layers called as MLPs. In an FC layer, all the output activations are the MAC results of the input activations. The FC layers require a compelling amount of memory and computation. The memory requirement for the FC layer of AlexNet [17] is 111.81 MB [30] [57] and similarly for GoogleNet [25] it is 1.95 MB [30] [57]. In low memory-centric applications like mobile phones, some connections between the activations are removed by setting the weight to zero to avoid unwanted memory access and MAC computations. Such types of layers are called sparsely-connected layers [29] [58].

DNNs can also be made computationally efficient by reducing the weights that do not contribute to the output. The weights are decreased mainly during the training by compression, which is extensively explained in chapter 5. Weight compression reduces the number of floating operations inside the DNN model. By compressing weight, full-connected layers can generate fewer floating-point operations [32] [59] [60]. The impact of weight reduction is explained in Section 5.3 of Chapter 5. The sparsity structure also emerges if each output is a function of a fixed size of inputs. The hardware can be made more efficient using the same set of weights in all computations. Iteratively using the same weights is known as weight sharing, which reduces overall weight storage needs.

A weight-shared DNN is constructed by structuring the computation as convolution, as illustrated in Fig. 2.9(b). Input activations are used to calculate the MAC sum for each output activation, and the same weights are used for all outputs. This type of layer is called a convolution layer.

2.6.1 Convolutional Neural Networks (CNNs)

The most common form of DNNs is the Convolutional Neural Network (CNN), which consists of multiple layers as shown in Fig. 2.10 and Fig. 2.11. In these networks, each layer continuously generates a higher level of abstraction of the input data called a feature map (fmap), which conserves essential and unique information. The modern CNNs can achieve a higher level of performance by employing a very deep hierarchy of multiple layers [61]. CNNs have a wide variety of applications, such as image classification, speech recognition, and robotics [27] [29] [62].

Most of the convolution layers in the CNN network are composed of high dimensional convolutions, as shown in Fig. 2.12. In the computation, the input activations of a layer are represented by a set of 2-D Ifmaps, each of which is called a channel. Each channel is convoluted with specific 2-D filters from the stack of filters, one for each channel. This stack of 2-D filters is referred to as a single 3-D filter. The convolution results of three different input channels

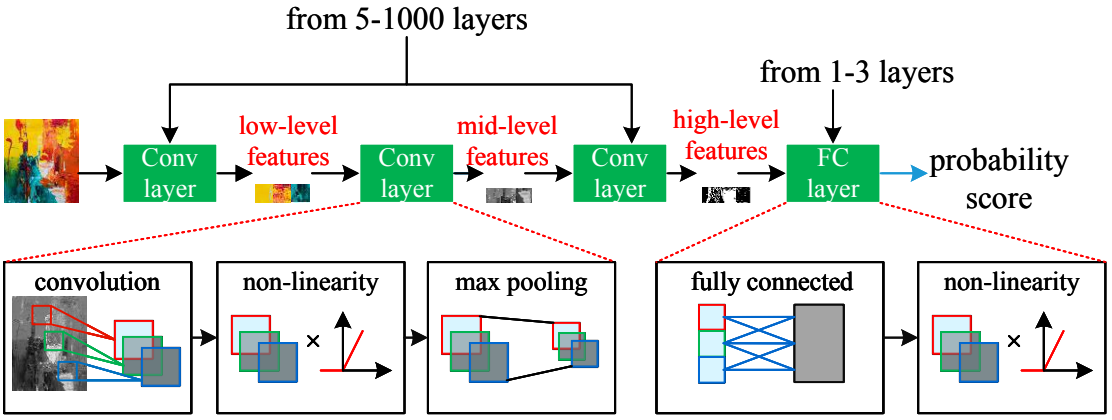


Figure 2.10: Convolutional neural networks.

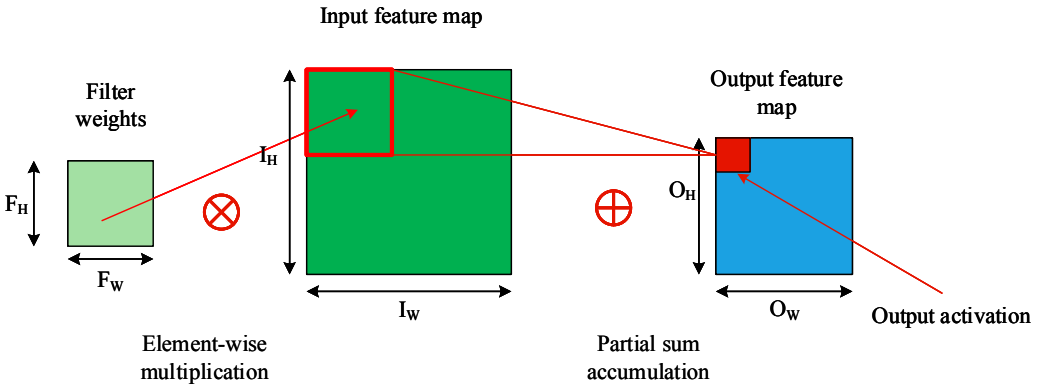


Figure 2.11: 2-D convolution.

Table 2.1: Parameters of the CONV Layer.

Shape Parameter	Description
N	Batch size of 3-D feature maps
O_C	Number of channels in the Output Feature Maps (Ofmaps)
I_C	Number of channels in the Input Feature Maps (Ifmaps)
F_C	Number of channels in the filter
O_H	Height of the Ofmaps
O_W	Width of the Ofmaps
I_H	Height of the Ifmaps
I_W	Width of the Ifmaps
F_H	Height of the filter weights
F_W	Width of the filter weights

with three filters are combined and added to the bias data. More recent networks [63] [64] have removed the usage of bias from certain parts of the layers. The computation results are the final output activations consisting of one channel Ofmaps explained below. The 3-D filters

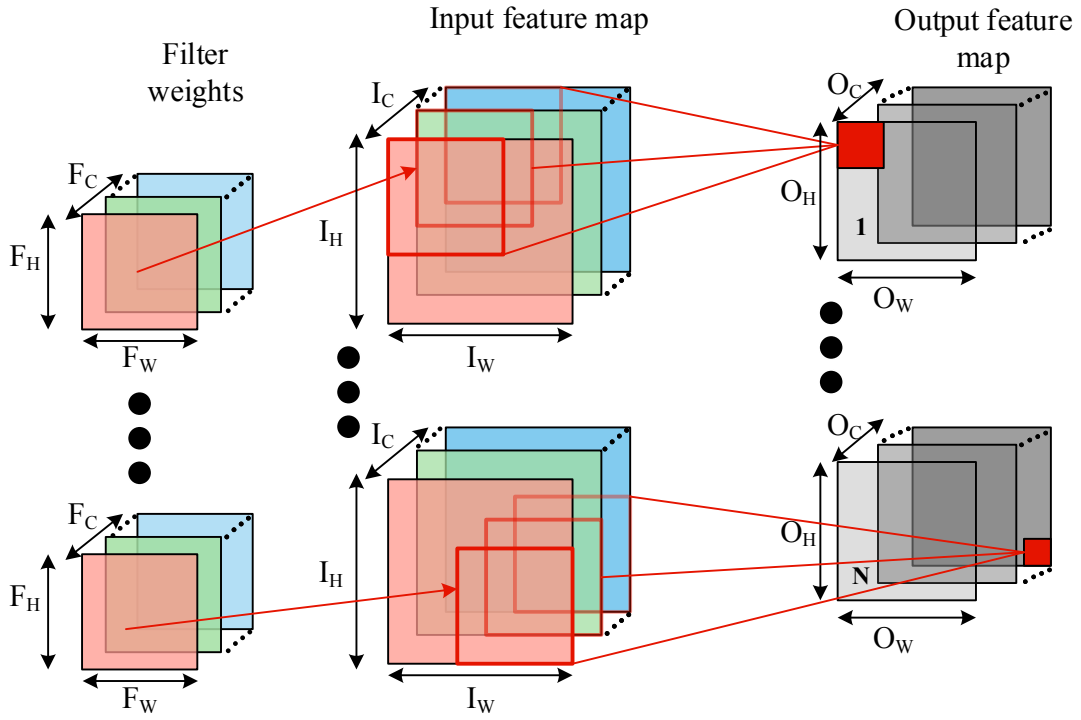


Figure 2.12: Multi dimensional convolution.

can be used on the same input to create multiple (or added) output channels. In addition, numerous Ifmaps are processed together as a batch to improve the reuse of filter weights.

The shape parameters with the description are shown in Table 2.1, and the computation of a convolution layer is defined in Eq. 2.6. I, W, B, and O are the matrices of the maps, filters, biases, and Ofmaps respectively. Finally, S is the stride size for the convolution operation [29].

$$\begin{aligned} O[z][u][x][y] = B[u] + \sum_{k=0}^{F_C-1} \sum_{i=0}^{F_W-1} \sum_{j=0}^{F_H-1} I[z][k][Sx+i][Sy+j] \times W[u][k][i][j] \\ 0 \leq z \leq N, 0 \leq u \leq O_C, 0 \leq x \leq O_W, 0 \leq y \leq O_H \\ O_H = (I_H - F_H + S)/S, O_W = (I_W - F_W + S)/S \end{aligned} \tag{2.6}$$

In SoA, CNN layers typically consist of five to more than a thousand convolution layers. Normally, the FC layers used for classification comprise one to three layers. The FC layers apply the filter weights on the Ifmaps like the CONV layers; the filter sizes are of the same size as ifmaps, and hence, they do not possess the weight sharing property of CONV layers. For the FC layer computation, the Eq. 2.6 remains the same however there would be additional constraints on the shape parameters: $I_H = F_H$, $I_W = F_W$, $O_W = O_H = 1$ and $S = 1$. The other optional layers in DNN networks are non-linearity, pooling, and normalization [29]. The function and computation of these layers are:

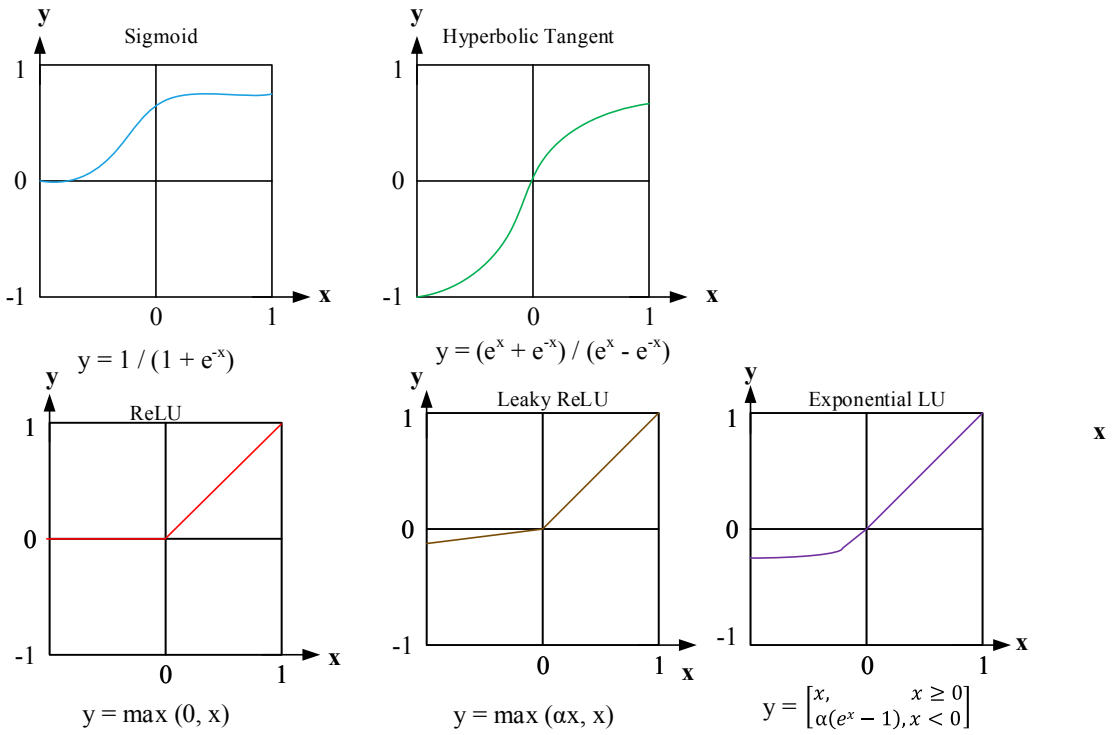


Figure 2.13: Different forms of non-linear activations [65].

2.6.1.1 Non-Linearity

Non-linear activation functions are typically used after the CONV or the FC operation. Different non-linear functions are used in DNN, as shown in Fig. 2.13. They include traditional non-linear functions such as sigmoid, hyperbolic tangent, and ReLU [66]. ReLU is one of the most popular models in the hardware AI community due to its simplicity, hardware friendliness, and ease of training. Variations of ReLU mainly include leaky ReLU [67], parametric ReLU [68] and exponential LU [69] for the increased accuracy. A non-linearity called maxout function [70] is used in the speech recognition tasks [71] to determine the maximum value of two intersecting linear functions. The pros and cons for each non-linear function are explained in Section 2.1.

2.6.1.2 Pooling

Pooling functions reduce the dimensions of the feature maps by selecting the highest magnitude number and eliminating the other ones. This function is applied separately to each channel, enabling the network to be more robust and invariant to minor shifts and other distortions. The primary need for pooling is the dimension reduction in the CNN network. Doing so reduces the number of parameters that must be learned from the network and its computation. It also helps summarize the features present in a region in the feature maps generated by the convolution layer. A set of values are pooled into the receptive field to fewer values. A simple example of a pooling (max and average) operation is shown in Fig. 2.14. A typical pooling consists of non-overlapping blocks, where the stride equals the pooling size. In most cases, the stride is more significant than one, which is used to reduce the dimensions of the feature maps.

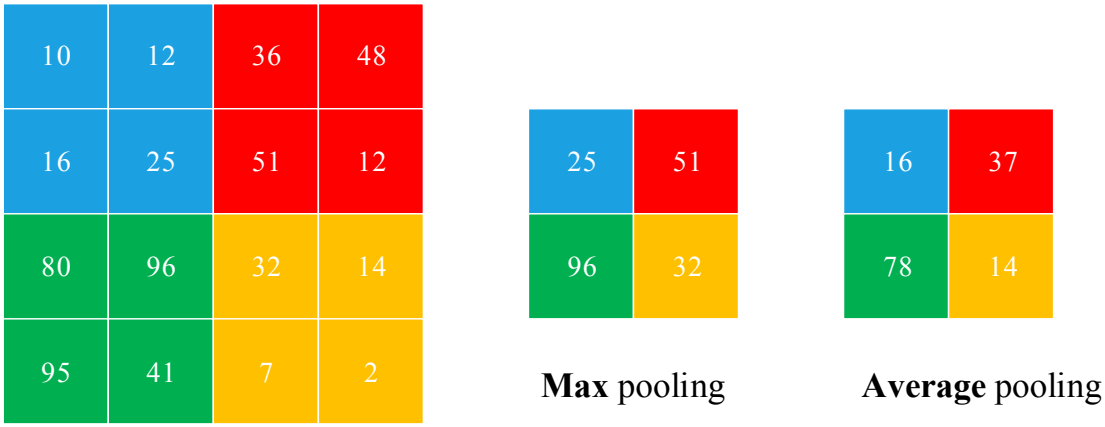


Figure 2.14: Different forms of pooling [65].

2.6.1.3 Normalization

Controlling the input distribution across the layers can be significantly improved by training speed and accuracy. First, the input activations are distributed across the layer such that it has a zero mean and a unit standard deviation. Then, the normalization value is scaled and shifted in batch normalization (BN) as shown in Eq. 2.7. Here, x represents the normalized input activations, and the parameters (γ, β) are learned during the training process. In addition, adding ϵ to the denominator provides numerical stability and avoids the drawback of numerical inefficiencies. Earlier to the use of BN, local response normalization (LRN) [17] was used, which was mainly inspired by lateral inhibition in neurobiology. In the LRN technique, higher-value activation subdued the lower-value activations. However, in recent times, for CNN training, BN is used in more prominence when compared to LRN because it makes the network more stable during training. It also aids in faster learning rates, which aids in the quicker learning process. In addition, BN is also performed between the CONV or FC layers and the non-linear function, while LRN is performed after the non-linear function.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.7}$$

2.6.2 Popular DNN Models

Several DNN models have been developed in the last two decades and are extremely popular today. Each model has a unique network architecture regarding layer shapes (number of channels, filter size, and filters), number of layers, layer types, and connections between layers. Knowing these variations and trends is essential for building efficient DNN models. In this section, we provide a brief overview of popular DNNs like LeNet [42] as well as those that won the ImageNet Challenge, as shown in Fig. 2.8. Table 2.2 demonstrates the different CNN models' summary.

LeNet [42] was the first CNN model introduced in 1989. It was designed and developed for handwritten digit classification for a grayscale image of size 28 × 28. The most prevalent version, LeNet-5, has two CONV and two FC layers. The filter size of each CONV layer is 5 × 5 (1 filter per channel), with six filters on the first layer and 16 filters on the second layer. A

Table 2.2: Summary of different CNN models.

Parameters	LeNet5	AlexNet	Overfeat	VGG 16	GoogleNet	ResNet 50
Top-5 accuracy	N/A	16.4	14.2	7.4	6.7	5.3
Input size	28×28	227×227	231×231	224×224	224×224	224×224
# of CONV layers	2	5	5	13	57	53
Depth in # of CONV layers	2	5	5	13	21	49
Filter sizes	5	3,5,11	3,5,11	3	1,3,5,7	1,3,7
# of channels	1-20	3-256	3-1024	3-512	3-832	3-2048
# of filters	20-50	96-384	96-1024	64-512	16-384	64-2048
Stride	1	1,4	1,4	1	1,2	1,2
Weights	2.6K	2.3M	16M	14.7M	6.0M	23.5M
MACs	283K	666M	2.67G	15.3G	1.43G	3.86G
# of FC layers	2	3	3	3	1	1
Filter sizes	1,4	1,6	1,6,12	1,7	1	1
# of channels	50,500	256-4096	1024-4096	512-4096	1024	2048
# of filters	10,500	1000-4096	1000-4096	1000-4096	1000	1000
Weights	58K	58.6M	130M	124M	1M	2M
MACs	58K	58.6M	130M	124M	1M	2M
Total weights	60K	61M	146M	138M	7M	25.5M
Total MACs	341K	724M	2.8G	15.5G	1.43G	3.9G

larger filter size is chosen for the first two layers to reduce the matrix dimensions. An average pooling mechanism of size 2 × 2 after each convolution is used instead because smaller filters are faster to compute. In addition, the sigmoid function is used as a non-linearity function. LeNet has around 60K weights and 341K MACs/image. To fit such large weights and images, we use the external DRAM, and a FIFO technique is used in the inference to fit the data into memory-constrained SRAM. CNN's first industrial success was LeNet, which ATMs used to recognize digits for cheque deposits. The main advantage of LeNet is that it is straightforward to implement and can be used for small applications like character recognition. In contrast, the disadvantage is that it is built only for black and white images, and the models are not deep, often leading to mediocre performance.

Among the CNN models, AlexNet [17] won the ImageNet Challenge in 2012 to resolve the vanishing gradient problem without restricting gradient values. AlexNet was the first GPU-based model and was faster than LeNet, as LeNet consisted of 60K training parameters, while AlexNet had 60 million training parameters. AlexNet possesses 5 CONV layers and 3 FC layers. The CONV layer has filters ranging from 96 to 384, and the filter size ranges from 3 x 3 to 11 x11, with the 3 channels of each of the 3 channels of the filter from the first layer corresponding to the RGB components of the image. The ReLU non-linearity function is used in this network. A max pooling of stride size 3 × 3 is applied to the outputs of layers 1, 2, and 5. A stride size of 4 reduces computation at the first computation layer. AlexNet differs from LeNet in terms of the number of weights and the shapes from layer to layer. For the further reduction in the computation in the second CONV layer, 96 output channels from the first layer are split into two groups of 48 input channels for the second layer, such that the filters in the second layer have 48 channels. In the same way, the weights for the fourth and fifth layers are split into two groups, respectively. AlexNet has 61M weights and 724M MACs to classify the 227 × 227 input image. The main advantage of AlexNet was that it was deeper than the other models introduced in 2012. Furthermore, it successfully negated the negative output summation of gradients without negating the dataset. Thus, the training speed was

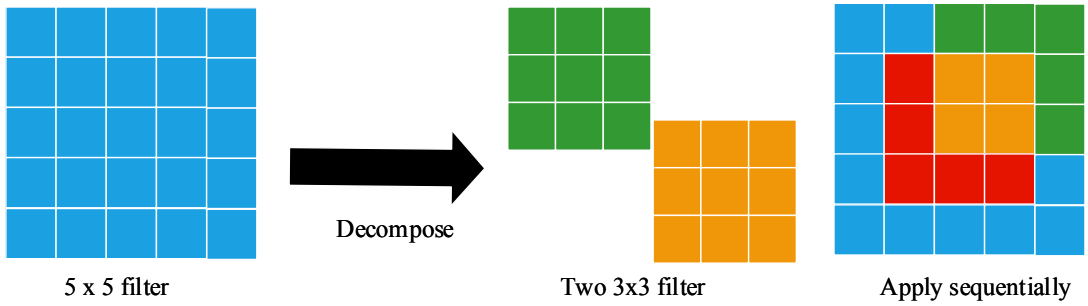


Figure 2.15: 5×5 constructed from 3×3 filter in VGG-16.

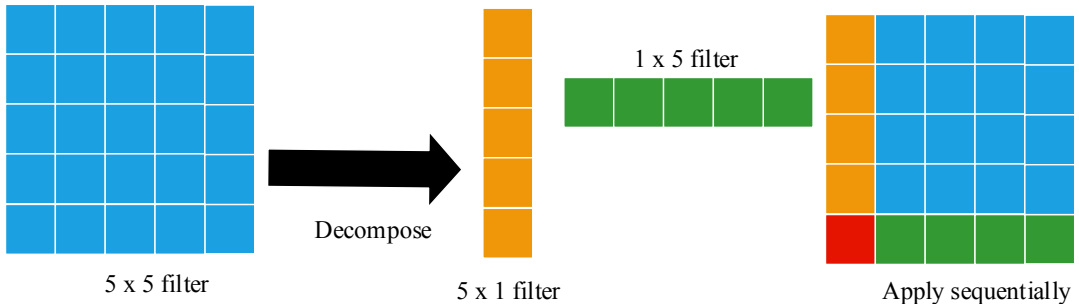


Figure 2.16: 5×5 constructed from 1×5 filter and 5×1 filter in GoogleNet.

improved since all perceptrons were not active. However, the disadvantage of this model is that it takes longer to reach higher accuracy.

Overfeat [72] is similar to AlexNet with 5 CONV layers and 3 FC layers. Overfeat architecture demonstrated the implementation of object localization, which AlexNet could not do. The main architectural differences are that the number of filters is increased from 384 to 512 in layer 3, 384 to 1024 in layer 4, and 256 to 1024 in layer 5. In addition, layer 2 is split into two groups, FC layer has 3072 channels instead of 4096, and the input size is 231 × 231 rather than 227 × 227. The total number of weights is 146M, and the total MACs are up to 2.8G/image. Overfeat has two different models, namely fast and accurate. The accurate model used in the ImageNet challenge has a 0.65% lower error rate than the fast model, but it comes at the cost of 1.9×, a more significant number of MACs. The main advantage is that it can perform object localization, while its disadvantage could be more profound, and it takes more time to achieve higher accuracy than AlexNet.

VGG-16 [44] has deeper layers when compared to AlexNet and Overfeat, which improved the overall training speed. It has 13 CONV layers and 3 FC layers. Larger filters in the range of 5×5 are built using multiple 3×3 filters with fewer weights to balance the cost of going more profound, as shown in Fig. 2.15. Hence, most of the CONV layers have a filter size 3×3. VGG-16 uses 15.5G MACs to process 224×224 images with 138M weights. VGG has two models, namely VGG-16 and VGG-19. VGG-19 has a 0.1% lower error rate than VGG-16 at 1.27× MACs. This model has a more significant non-linearity with more small kernels, which helps reduce negative values more quickly. However, it suffered from a vanishing gradient problem and was smaller than the ResNet model.

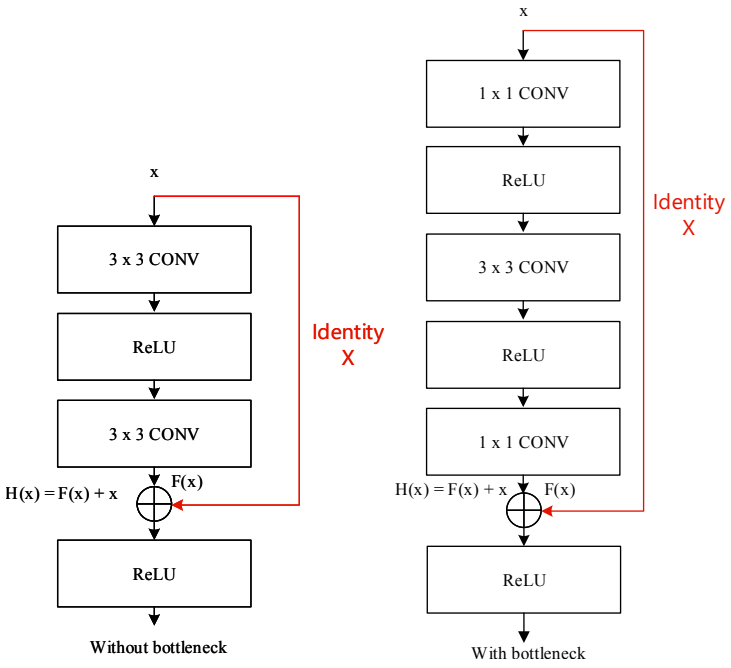


Figure 2.17: ResNet with and without bottleneck.

GoogleNet [25] is much more profound with 22 layers, and it also introduced the inception module, as shown in Fig. 2.16. The inception module has parallel connections, unlike the previous single serial connection. Unique filter combinations (1×1, 3×3, 5×5) are used for each parallel connection, and their outputs are concatenated for the module output. Multiple filter sizes affect the input processing at various levels. For efficient training, GoogleNet stores weights and activations as a backpropagation during the training phase that could fit into the entire GPU memory. To reduce the number of weights, 1×1 filters are used as the bottleneck to minimize each filter's total number of channels. The 22 layers have 3 CONV layers, 9 inception layers, and one FC layer. Since its development, GoogleNet has 3 versions: v1, v3, and v4. Inception-v3 decomposes the convolutions using the smaller 1-D filters to reduce the number of MACs and weights to 42 layers. In incoherence with batch normalization, v3 achieves a 3% lower top-5 accuracy than v1 with a 2.5× increase in computation. The main advantage is due to the deeper layers of the GoogleNet; it trains faster. The pre-trained GoogleNet is 96MB, comparatively smaller than VGG-16, which is 500MB. Even though there are no significant disadvantages, there is a divergence limit in the inception module called the Xception network.

ResNet [16] is also known as Residual Net and is a very deep network with more than 34 layers. ResNet was the first DNN network in the ImageNet challenge that exceeded human accuracy with a top-5 error rate of less than 5%

MobileNet [73] is based on a factorized convolution model called the depthwise separable convolution model. This DNN factorizes the standard convolution procedure into depthwise convolution and 1 × 1 convolution called pointwise. The depthwise convolution applies a single filter to each of the input channels. In contrast, the pointwise convolution applies 1 × 1 convolution to combine the outputs of the depthwise convolution as shown in Fig. 2.18. In MobileNet, all the layers are followed by a BN and ReLU as the non-linear activation

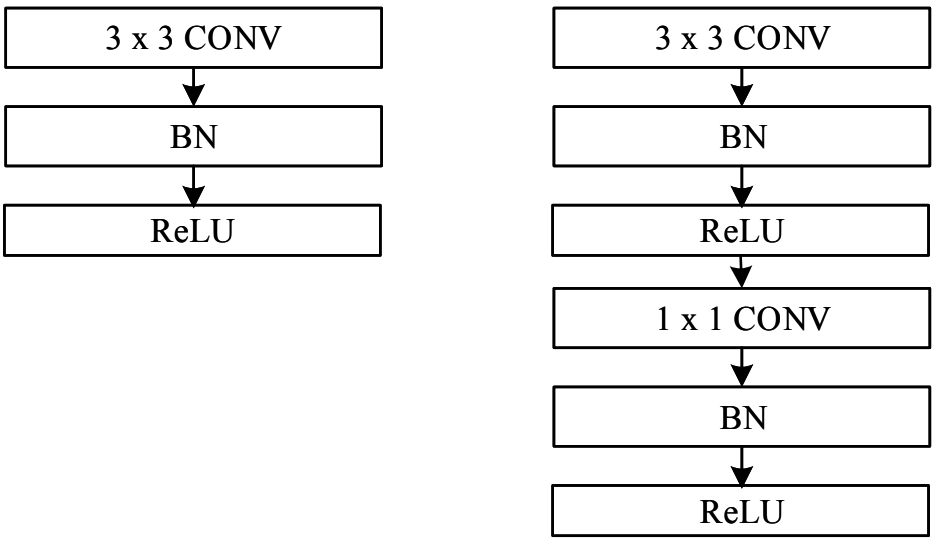


Figure 2.18: MobileNet, the left side indicates a standard convolutional layer with BN and ReLU. The right side shows depth-wise separable convolutions with depthwise and pointwise layers followed by BN and ReLU.

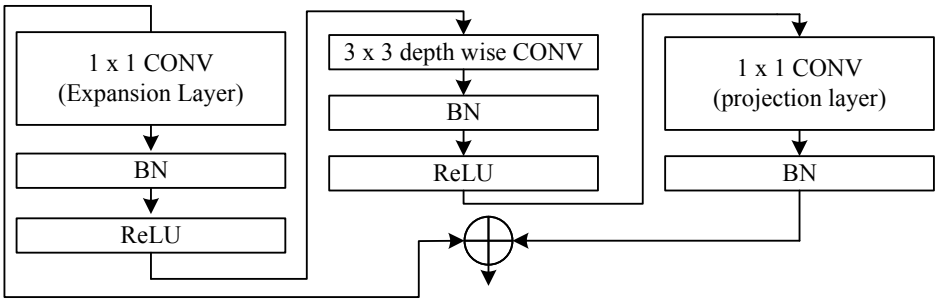


Figure 2.19: MobileNet v2 network.

function with the one FC layer and softmax to perform the classification. MobileNet provides a top-5 accuracy of 89.9% with 4.24M parameters and 569M MACs/image. MobileNet has three versions, namely: MobileNetV1, MobileNetV2, and MobileNetV3. The structure of the MobileNetV1 is as explained in Fig. 2.18. MobileNet V2 [74] builds upon the concepts from MobileNet V1. However, it introduces two new features into its architecture: linear bottlenecks between each layer and shortcut connections between the bottlenecks, as shown in Fig. 2.19. In the V2 version, the pointwise convolution number of channels is smaller than in the V1 version. Hence, one of the layers is known as the projection layer, which projects the data with higher dimensions into a tensor with much lower dimensions. In addition, the V2 version supports residual connections like the ResNet. The MobileNet V2 consists of 53 layers, followed by a classification layer. It has a total of 3.47M parameters and a top-5 accuracy of 91%. MobileNet V3 [75] is the definitive version of the MobileNet series. In this version, small filters like 3×3 were replaced by 16×16 to provide good speedup in the computation, and the non-linear activation function ReLU is replaced by the Swish function according to Eq. 2.8. MobileNet V3 provides a total of 92.2% top-5 accuracy with 2.9 M parameters. The main advantage of all the MobileNet architectures is that it has reduced network size of around 17MB, reduced parameters of 4.2 million, and faster performance that helps to fit the mobile

Table 2.3: Different models of EfficientNet with weight parameters and top-5 accuracy results.

Model	Top-5 accuracy	# of parameters
EfficientNet-B0	93.30%	5.3M
EfficientNet-B1	94.40%	7.8M
EfficientNet-B2	94.90%	9.2M
EfficientNet-B3	95.70%	12M
EfficientNet-B4	96.40%	19M
EfficientNet-B5	96.70%	30M
EfficientNet-B6	96.80%	43M
EfficientNet-B7	97.00%	66M

devices. There is only a slight disadvantage for the MobileNet: the slight reduction in the accuracy compared to other SoA DNN models.

$$f(x) = x \cdot sigmoid(x) \tag{2.8}$$

SqueezeNet [52] was developed to fit the DNN on low-memory rigid platforms like the FPGAs. SqueezeNet starts with a standalone convolution layer CONV1, and 8 fire modules follow it and ends with the last CONV layer CONV10. The number of filter dimensions is increased for each fire module, and max pooling is employed to reduce the dimensions of the PSums. As a result, SqueezeNet has a 50%

- SqueezeNet has more aggressive channel reduction by consolidating the two-stage squeeze module.
- A separate 3 × 3 convolutions are used to reduce the model size and remove the added 1 × 1 branch after the squeeze model.
- Residual connection in the element-wise addition is similar to ResNet, which allows the network to be trained deeper without much of the vanishing gradient problem.
- Optimizing the baseline SqueezeNet architecture by simulating the performance on a multi-processor embedded system.

Overall, SqueezeNet uses around 1.5M weight parameters and provides a top-5 accuracy of around 88.80%. Its drawbacks are low classification accuracy and high computational complexity.

EfficientNet [76] uniformly scales the depth, width, and resolution using a simple and effective technique called compound efficiency. This network proves that the network saturates have higher depth and width accuracy. Compared to the other DNNs, EfficientNet is 8.4x smaller and 6.4x faster on inferences. EfficientNet has seven different versions. Table 2.3 shows the top-5 accuracy with the total number of parameters.

2.7 Different DNN Development Resources

The abundance of free development resources available from the academic community and industry contributed to the rapid development of DNNs. These resources play a pivotal role in developing DNN accelerators by providing characterizations of the different workloads and aiding the exploration of trade-offs in terms of accuracy and model complexity.

2.7.1 Frameworks

Many deep learning frameworks have been developed from various sources to facilitate the development of DNNs and enable the sharing of trained networks. The open-source libraries have a large number of software libraries for the DNNs. For example, developed at UC Berkeley in 2014, Caffe supports C, C++, Python, and MATLAB. Google released TensorFlow in 2015, which was created in C++ and Python. TensorFlow supports multiple CPUs and GPUs and is more flexible when compared to Caffe. The computation flow is expressed as dataflow graphs to manage the multidimensional arrays. Facebook developed a torch framework that supports C, C++, and Lua. Other higher-level libraries run on top of the frameworks above for efficient and faster development with a more universal experience. One example is Keras libraries, which are coded in Python and support other frameworks, such as TensorFlow, CNTK, and Theano.

These frameworks are one convenient aid for DNN researchers and model designers. In addition, these frameworks are invaluable for the efficient processing of the DNN engines. Also, these frameworks perform large primitive operations like CONV. Moreover, they make use of optimized software and hardware accelerators. This kind of acceleration is made clear to the framework user. For example, most of the frameworks use Nvidia's cuDNN library for fast execution of the Nvidia GPUs. Similarly, transparent incorporation of dedicated hardware accelerators can be achieved with the multi-bit accelerator [31], and FantastIC4 [32].

These frameworks are a beneficial source of workloads for most hardware researchers. They help them drive the experimental designs for different workloads and explore the hardware-software trade-offs.

2.7.2 Prominent Datasets for the Classification Application

When comparing DNN models, one of the most important things to consider is how each model oversees the task's difficulty. When comparing DNN models, one of the most important things to consider is how each model manages the task's difficulty. When comparing DNN models, one of the most important things to consider is how each model handles the task's difficulty. For example, classifying the handwritten digits from the MNIST dataset is much easier than classifying an object from a thousand different classes like the ImageNet dataset. For most complex tasks, the DNNs and the MACs size will be 10× more than the simple tasks. Also, the overall throughput will be lower with high energy consumption. LeNet-5 is used for handwritten digit recognition, while AlexNet, MobileNet, GoogleNet, and ResNet are used for image classification.

Most AI projects are developed using the freely available datasets in the market. These freely available datasets are essential for comparing the accuracy of different training approaches.

Image classification tasks are the most straightforward as they classify one complete image with a thousand different object classes for which the image belongs to [17] [29].

MNIST dataset was the most widely used dataset for handwritten digit classification. The grayscale image in the dataset is 28 × 28 pixels. The dataset possesses 10 different object classes, 60,000 training images, and 10,000 test images. LeNet-5 was the first neural network to use the MNIST dataset with an accuracy of 99.05%. Since then, accuracy has been increased to 99.79% using regularization of neural networks with drop connect [77]. The MNIST dataset is also considered one of the early datasets [78].

CIFAR [79] is a dataset comprising 28 × 28 pixels colored images and a subset of 80 million image datasets. CIFAR has 10 different object classes, 50,000 training images, and 10,000 test images. A two-layer convolutional DBN achieved a total accuracy of 64.84% on CIFAR-10. Since then, due to the development of fractional max-pooling, the accuracy has been increased to 96.53%.

ImageNet [80] is the largest dataset that predominantly works with a colored image of size 256 × 256 with 1000 different object classes. The object classes are defined using WordNet as a determining factor to combine synonyms into the same object category. Each object class has a hierarchy for the ImageNet groups. The 1000 classes are selected so there is no overlap in the ImageNet hierarchy. The ImageNet dataset has many fine-grained categories, including 100 different classes of plants. In addition, there are almost 1.3M training images, 100K test images, and 50,000 validation images.

The accuracies for the ImageNet challenges are categorized using two metrics, namely, top-5 error and top-1 error. The top-5 error means that if one among the top-5 object classes belongs to the input image, it is considered a correct classification. The top-1 requires the object class with the highest probability score to match the input image. In 2012, AlexNet won the ImageNet challenge with 83.6% top-5 accuracy, and the second-highest was 73.8%. The top-1 accuracy of the AlexNet was 61.9%. In 2017, the highest top-5 accuracy was 98.8% from the EfficientNet-L2 model.

2.7.3 Datasets for Other Tasks

As most DNNs have surpassed human accuracy in image classification, the ImageNet challenge has started focusing on more complex tasks such as single object localization and object detection. In the single object localization tasks, the target object must be classified and localized out of 1000 classes. DNNs generate the top-5 bounding box locations and top-5 categories. All the objects will be localized and classified among the 200 classes for object detection. The bounding box for all the objects belonging to a particular category must be labeled, and the objects not labeled are fined and removed as duplicated detections.

Other datasets are much more powerful and robust than ImageNet for computer vision tasks. For example, the PASCAL VOC dataset is extensively used for object detection, and it has 11,000 images representing around 20 classes. Similarly, MS COCO has 2.5M instances with 328K images for object segmentation, detection, and recognition. For precise 2-D localization, MS COCO has fewer categories but more instances per category. To aid the contextual information, MS COCO also has more labeled instances per image.

Presently, they are larger datasets that are made publicly. For example, the YouTube dataset has 8M videos covering over 4800 classes, and the Google image dataset has over 9M images holding more than 6000 categories. These large datasets are important as DNNs go deeper and have more significant weight parameters to train. Moreover, considering the more complex problems to solve shortly, both the large datasets and datasets with the new domain are essential for exploring the efficiency of the DNN engines.

2.8 Summary

DNNs have been essential in the technological world in the past decade. AI applications such as image and video recognition, speech and natural language processing, computer vision, and robotics are extensively used and often perform better than humans. This chapter provided in-depth information on the background of DNNs, different layers of DNNs, the most popular DNN models, and different frameworks and datasets involved in developing these models.

3

Hardware Techniques for DNN Inference

3.1 Introduction

Due to the increased popularity of DNNs in various applications, many hardware techniques have unique characteristics that allow for the targeting of DNN inference. For example, the Intel Knights Landing CPU has unique features like special vector instruction sets for deep learning. The Nvidia GPU PASCAL GP100 has 16-bit floating-point arithmetic support to perform two floating-point operations on a single-precision core for faster deep-learning executions. Exclusive systems especially are built for DNN processing tasks, such as Nvidia DGX-1 and Facebook's Big Basin custom DNN server [81]. Big Basin can train models with high efficiency, from 7T flops to 10.6T flops. DNN processing has been demonstrated on various SoC platforms like Nvidia Tegra, Samsung Exynos, and other FPGAs. Hence, it is essential to know the processing in these platforms and how ASIC accelerators are designed for DNNs to improve throughput and energy efficiency.

The MAC operations are the standard processing element in both the CONV and FC layer processing, which can be efficiently parallelized. Both temporal and spatial architectures are used to achieve high-performance throughput. Both these architectures have identical structures, with the control element being centralized, as shown in Fig. 3.1. The temporal architecture is extensively used in CPUs and GPUs. This architecture mainly employs parallel computation techniques such as SIMD (Single Instruction Multiple Data) and SIMT (Single Instruction Multiple Thread). The temporal architecture uses a centralized control for many processing elements (PE). The PEs can fetch the data from the memory hierarchy and cannot directly communicate with each other.

In contrast, in spatial architecture, the PEs form a communication chain to pass the data directly from one another. Each PE inside the spatial architecture also possesses control logic and a scratchpad's local memory. Spatial architectures are used in computation-centric platforms like DNN processing in ALUs and FPGAs.

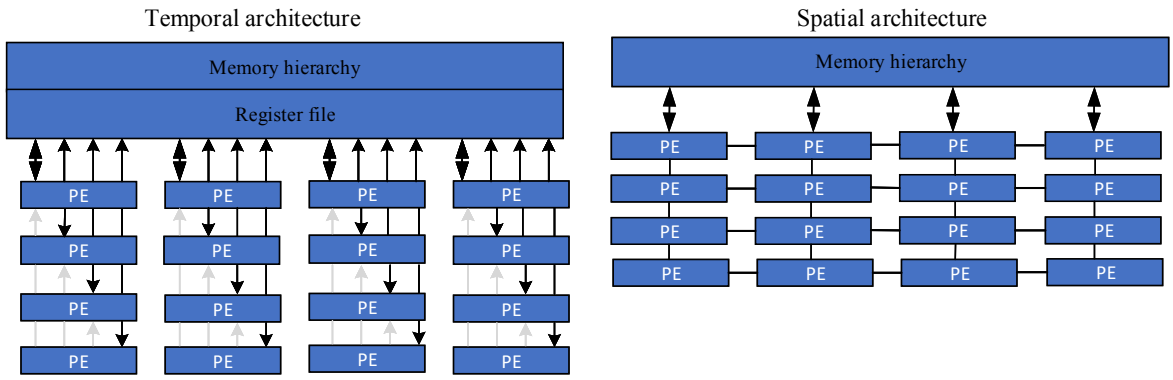


Figure 3.1: Parallel computing architectures [29]

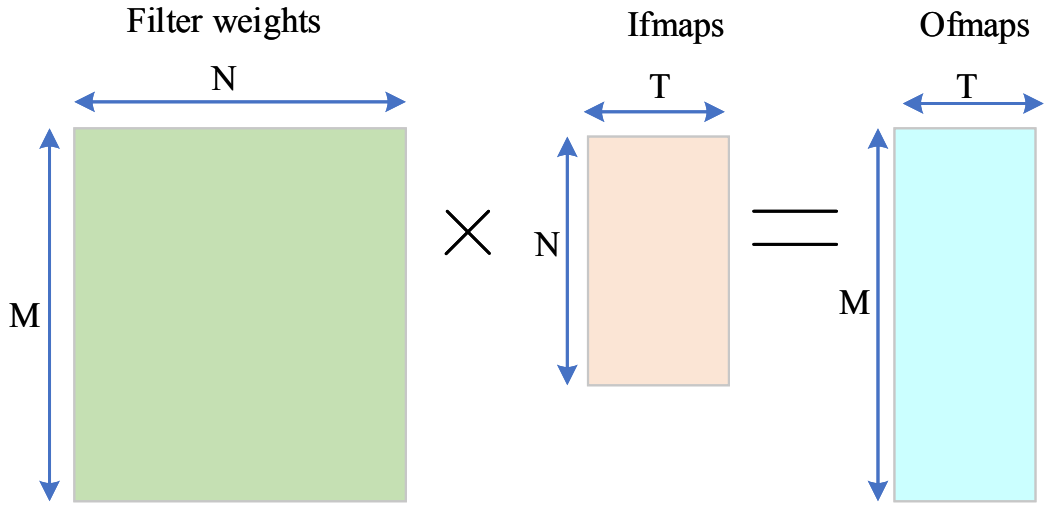


Figure 3.2: Matrix multiplication for fully connected layers [29].

3.2 Matrix Multiplication on CPU and GPU Platforms

Most of the CPUs and GPUs employ SIMD and SIMT parallelization techniques to perform MAC operations in parallel. The ALUs in the CPU and GPU use the same control, and the memory of the ALUs for the FC and CONV layers use matrix multiplication. Fig. 3.2 shows the matrix multiplication for the FC layer. The filter matrix height equals the number of filters (M), and the matrix width equals the number of weights per filter (N). The height of the Ifmaps is the total number of activations per Ifmaps (N), and the width is the total number of Ifmaps (T). The height of the Ofmaps is the total number of channels (M), while the width is the number of Ofmaps (T). Here, each Ofmaps of the FC layer has the dimension of $1 \times 1 \times$ number of output channels.

The CONV layer in DNN can be mapped to matrix multiplication using the Toeplitz matrix as shown in Fig. 3.3. The main downside of using matrix multiplication for the CONV layers is the redundant data in the Ifmaps. Most of the time, it leads to complex memory access or inefficiency during data storage.

Many software libraries are designed for CPUs like Intel MKL and Open BLAS and for GPUs like cuDNN and cuBLAS that optimize for matrix multiplications. However, matrix

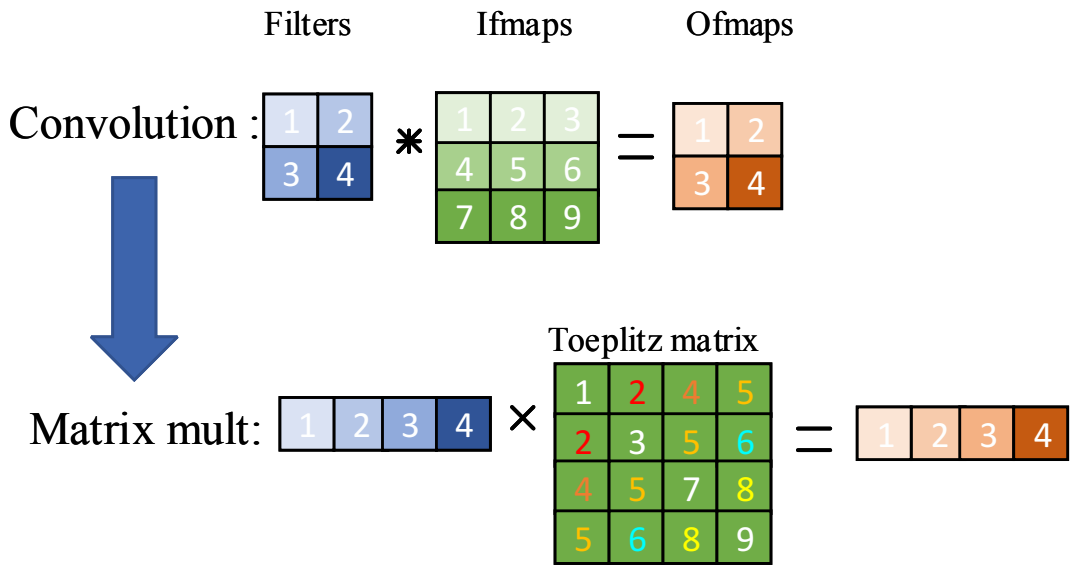


Figure 3.3: Matrix multiplication for convolution layers [29].

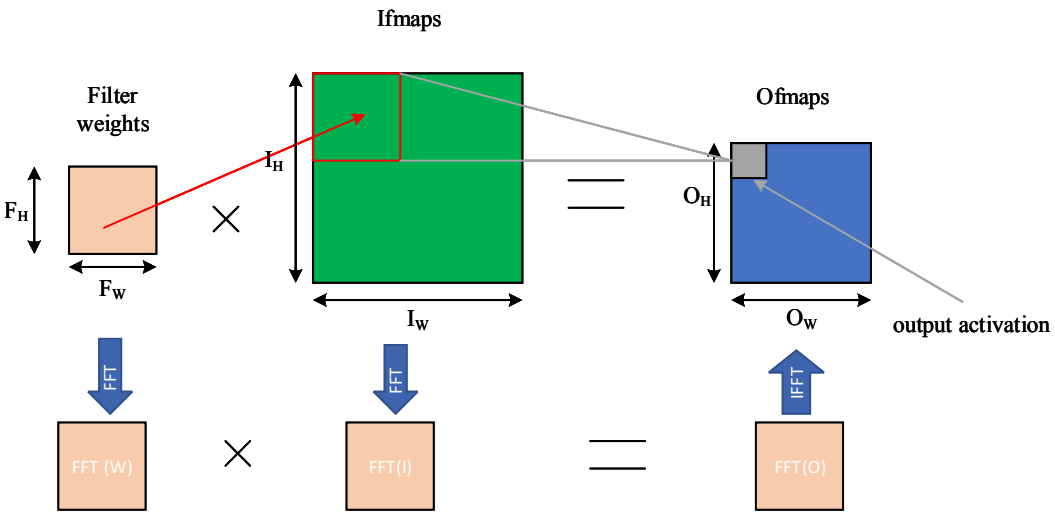


Figure 3.4: FFT for DNN processing [29].

multiplication is mainly tiled to the storage hierarchy for most of these platforms, which are ordered in a few MBs at the higher layers.

Applying computational transformations to the data can speed up CPUs and graphics cards that perform matrix multiplication while maintaining the same bit-wise accuracy. However, this comes at the cost of increased additions and irregular data access patterns.

The Fast Fourier Transform (FFT) [82] [83] is one of the popular approaches shown in Fig. 3.4. FFT reduces the total number of multiplications from $O(N_o^2 N_f^2)$ to $O(N_o^2 log_2 N_o)$, where the output size is $N_o \times N_o$, and the filter size is $N_f \times N_f$. For the convolution computation using the FFTs, we multiply filters and the Ifmaps in the frequency domain and apply an inverse FFT to the resulting output product to recover the output products in the time domain or the spatial domain. However, using the FFTs comes with significant drawbacks, such as:

1. Coefficients in the frequency domain are overly complex.

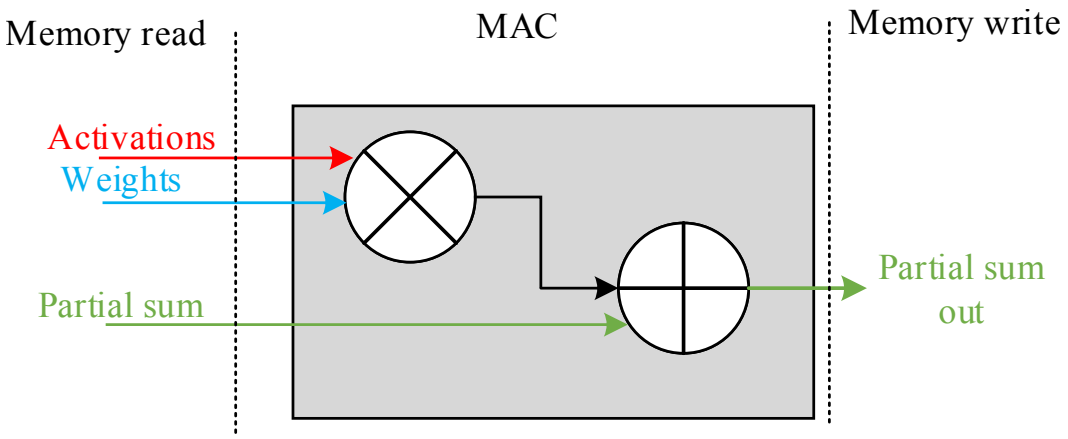


Figure 3.5: Read and write access per MAC [29].

2. As the filter size increases, the advantages of FFTs decrease.

3. Size of the Ofmaps is determined by the size of the FFT, which is much larger than the filter size.

Even though the FFT reduces the total computational requirement, it increases the memory storage capacity and the bandwidth demand.

Several optimization techniques have been proposed on the FFT to make computation effective for the DNNs. Pre-compiling and storing the FFT of the filter ahead of time reduces the total number of operations. The Ifmaps should be computed once and reused multiple times to generate multiple channels in the Ofmaps. Most of the data contain real values, and its corresponding Fourier Transform is symmetric, which can be exploited to reduce the overall storage and computation cost.

In practice, different algorithms have been used for different layers and sizes. For example, FFT uses 3×3 filters, Winograd algorithm uses 5×5 filters and Gaussian FFT uses $3 \times$ filters [84]. The existing platforms like MKL and cuDNN dynamically chose the appropriate algorithm for a given shape and size [85].

3.3 Different Dataflow Techniques for Energy Efficient DNN Accelerators

For the DNN inference, the biggest bottleneck is the memory access from the DRAM to the on-chip SRAM. Each MAC in the processing unit requires a minimum of three memory reads: filter weight fetch, activation fetch, and PSums. In addition, it requires one memory write for the updated PSums as shown in Fig. 3.5. In the worst-case scenario, all the memory access has to go through the off-chip DRAM, affecting both the throughput and the energy efficiency. For example, the AlexNet has 724M MACs operations, which require 3G DRAM access. In addition, most of the DRAM access requires several orders of more energy than the computation. The powerful DNN models are more significant and must be fetched for every image access [27]. As explained in [26], memory for neural networks is required to store input

Table 3.1: Energy requirement for memory and computation.

Operation	Energy Requirement
8-bit integer addition	0.015 pJ
32-bit integer addition	0.096 pJ
8-bit integer multiplication	0.196 pJ
32-bit integer multiplication	3.078 pJ
8-bit floating point addition	0.3108 pJ
32-bit floating point addition	0.9108 pJ
8-bit floating point multiplication	0.565 pJ
32-bit floating point multiplication	3.437 pJ
8KB cache memory access	8 pJ
32KB cache memory access	17 pJ
1MB cache memory access	102 pJ
DRAM memory access	1.5 - 2.8 nJ

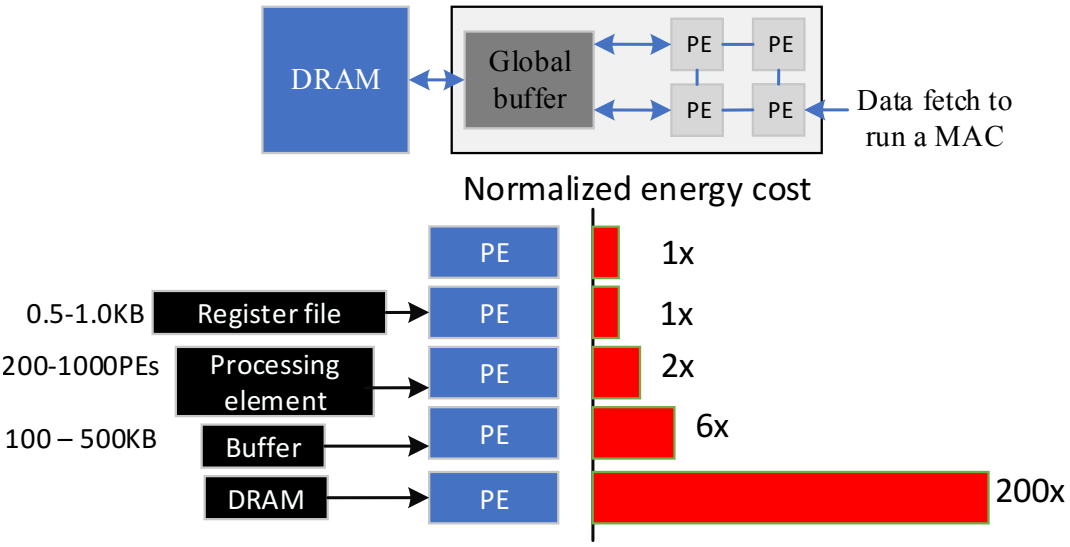


Figure 3.6: Energy consumption for different memory hierarchy [29].

activations, weight data, and corresponding output data of each convolution and FC layer. Fetching each layer from the external memory increases latency and power consumption since tens of MBs must access each layer. At the same time, on the computation side, the data are calculated on-chip without any external interference, thus reducing the power requirement and the energy consumption. The energy costs for the memory and computation are shown in Table 3.1.

Fig. 3.6 shows how spatial accelerators reduce energy consumption by introducing multiple levels of local memory hierarchy with varying energy costs. For example, in addition to the global buffer or scratchpad, an extensive inter-PE network, including a systolic array, can pass data directly between the PEs. Each PE contains a register file (RF) of a few KBs. Also, multiple memory hierarchy levels will help improve energy efficiency by providing efficient data access. For instance, fetching the data from the RF or the neighboring PEs would be most efficient.

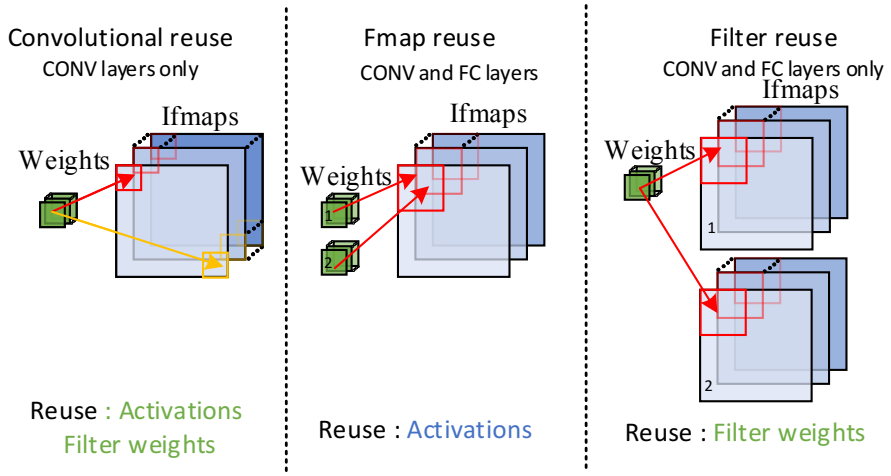


Figure 3.7: Data reuse strategy [29].

Most accelerators are designed to support a specialized data flow that uses the memory hierarchy. The data flow also decides what data needs to be read by the memory hierarchy and what needs to be written or processed by the memory hierarchy. To support the best energy efficiency, it is possible to maintain a fixed data flow that can adapt to various DNN shapes and sizes. The optimized data flow reduces the energy consumption from the memory access. Large memories consume more energy than compared to smaller memories. In contrast, on-chip SRAM stores kilobytes of data and consumes less power, while DRAM stores gigabytes. Data for SRAM is stored in transistor latches, whereas for DRAM, it is stored in capacitors [29]. Thus, whenever data is moved from an energy-expensive memory component to an energy-coefficient memory component, we need to reuse them as much as possible to reduce the energy access requirement. However, the SRAM has a limited storage requirement, and thus, the reuse factor must be explored as much as possible.

For the DNN processing, we mainly evaluate the different data flows that exploit the input data reuse like convolutional data reuse, feature map reuse, and filter reuse technique, as shown in Fig. 3.7. For convolutional reuse, both the Ifmaps activations and filter weights are reused for a particular channel. Similarly, for feature map reuse, multiple filters are reused for the same feature map so that the Ifmaps activations are reused multiple times across the filters. Finally, multiple Ifmaps are refined once for filter reuse, and the corresponding filter weights are used several times across the Ifmaps.

The three types of data reuse can be performed by storing the data in the local memory, such as a scratch pad, and reusing those multiple times without going through the DRAM. Also, for example, in AlexNet, the total DRAM reads can be reduced by 600× in the CONV layers if we are reusing the local memory instead of the DRAMs for storing the PSums. A local memory hierarchy can reduce DRAM access from 4200M to 62M [29] by reusing and accumulating data within the on-chip memory without requiring external memory.

The working operation of the DNN accelerators is similar to that of the general-purpose processors. In conventional processors, the compiler inside the processor translates the machine-readable binary codes to execute the hardware architecture. Similarly, in the processing of the DNNs, the mapper translates the DNN size and shapes into a hardware-readable computation

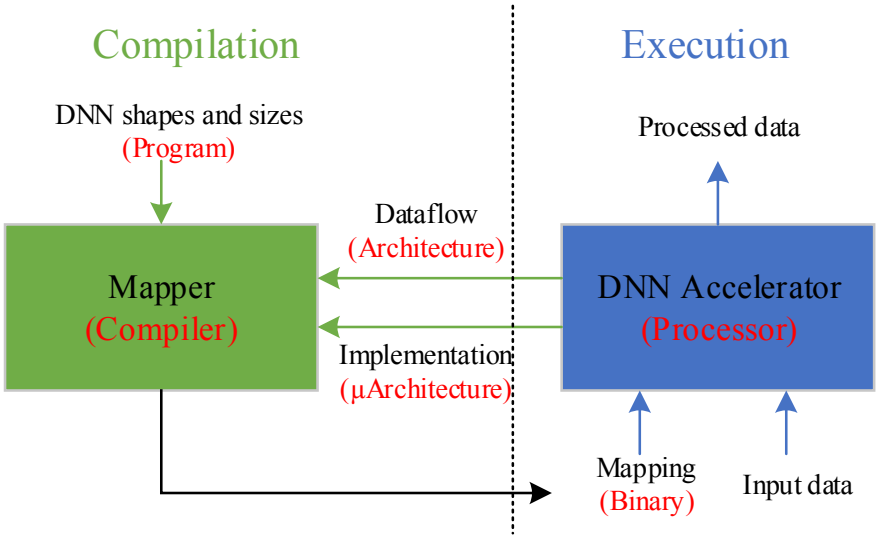


Figure 3.8: Analogy of operation between DNN accelerators (black texts) and general-purpose processors (red texts) [28].

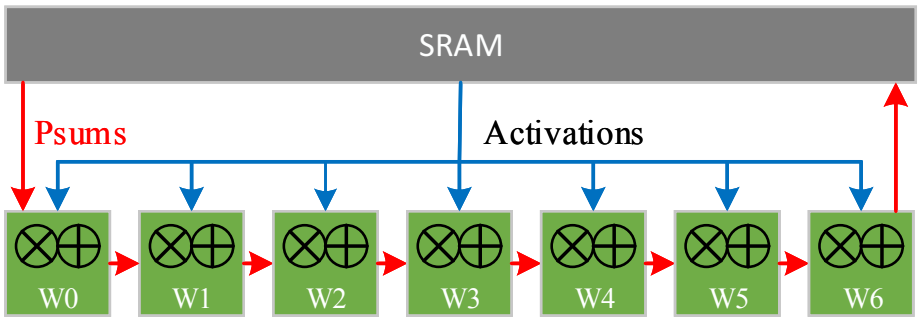


Figure 3.9: Weight stationary dataflow.

mapping for executing the given data flow. The compiler optimizes the hardware performance, and the mapper optimizes the energy efficiency. A simple example is shown in Fig. 3.8.

Different DNN dataflow techniques can be employed to improve data handling for efficient data processing.

3.3.1 Weight Stationary (WS) dataflow

In the stationary weight dataflow, weights are read from the DRAM while at the same time being accessed from the scratchpad (SP) or register file (RF) with the lowest energy consumption, as shown in Fig. 3.9. In most cases, each of the weights is read from the DRAM into the RF/SP for each PE, and it stays stationary for further processing. The computation unit performs as many MAC operations as possible for the exact weights while the weight is present in the RF/SP. Therefore, it maximizes the use of convolutional and filter weights. In the WS dataflow, the inputs and PSums move through the spatial array and the SRAM. In the first step, the Ifmaps activations are broadcast to all PEs. Then, the PSums are accumulated across the PE array.

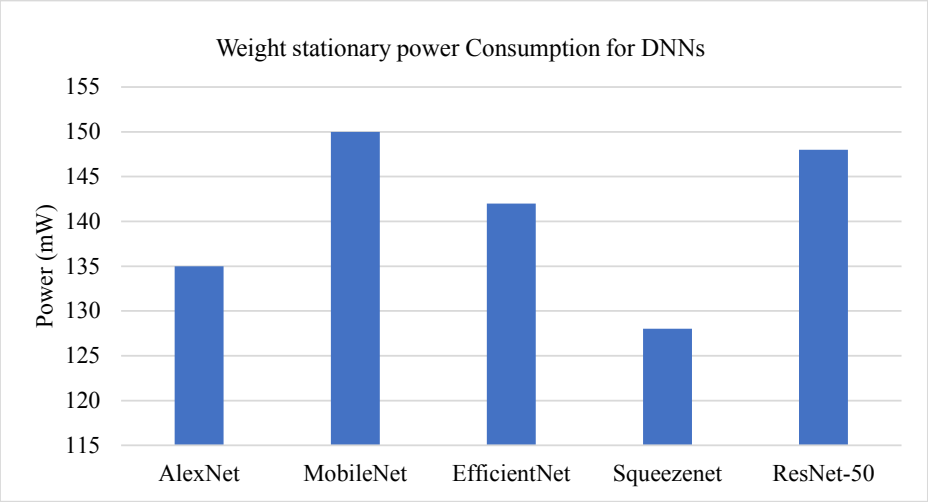


Figure 3.10: Power consumption for different neural networks using weight stationary dataflow.

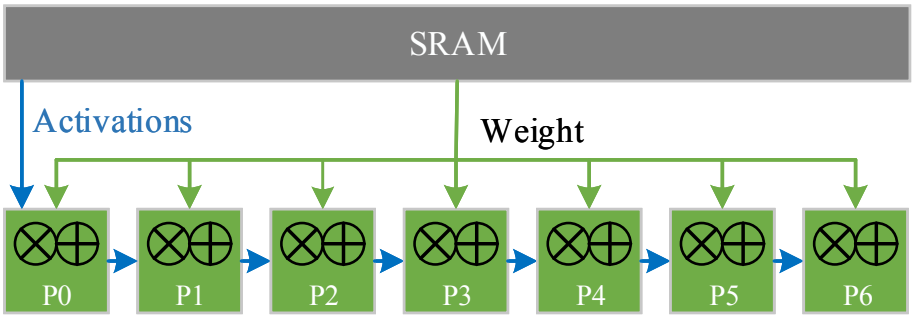


Figure 3.11: Output stationary dataflow.

One of the examples of previous work that implements weight static dataflow is NeuFlow [86], which employs eight 2-D CONV engines for processing a 10 × 10 filter. There are 100 MAC (PE) units, where the weight remains stationary for processing. The Ifmaps are broadcasted for all the MAC units, and the PSums are accumulated across the MAC units. Additional storage elements are added correctly to accumulate the PSums. The power consumption for different neural networks using a weight stationary dataflow is shown in Fig. 3.10.

3.3.2 Output Stationary (OS) dataflow

The output stationary dataflow helps reduce the energy consumption for reading and writing of the PSums. The accumulation of the PSums for the same output activation value is stored in the RF/SP Fig. 3.11. For the PSums accumulation inside the RF/SP, the input activations are streamed across the array of PEs, and the weights are broadcasted to all the PEs in the array.

The OS dataflow is implemented in the ShiDianNao processor, where each PE processes each output activation value by fetching the input activations from the adjacent PEs. The dedicated network is implemented across the PE array to pass the data horizontally and vertically. In addition, each PE has a data delay register (typically a D-flip flop) to store the data for a required number of cycles. At a system level, the SRAM streams the input

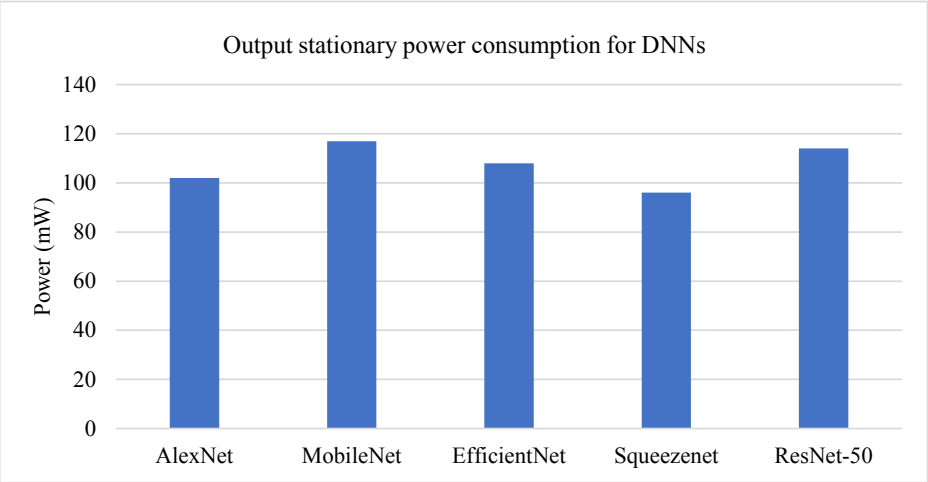


Figure 3.12: Power consumption for different neural networks using output stationary dataflow.

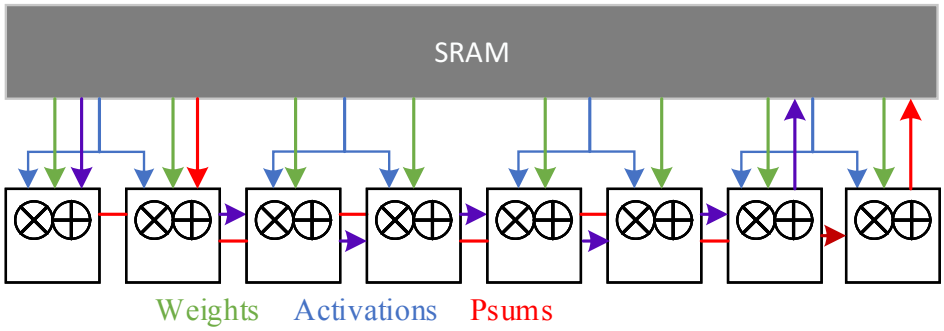


Figure 3.13: No local reuse dataflow.

activations and broadcasts the weights across the PE array. The PSums are accumulated inside each PE and then streamed out back to the PSums memory.

OS dataflow can be implemented for different variants, as the processed output activations come from other dimensions. For example, one variation targeting the CONV layers has a single output channel and multiple-output activations; the CONV layer processing is mainly performed for the output activations to maximize the data reuse. Similarly, for processing the FC layers, the variants focus on generating the output activations for different channels, and each channel has only one output activation. The power consumption for different neural networks using output stationary dataflow is shown in Fig. 3.12.

3.3.3 No local Reuse (NLR) dataflow

In some instances, small RF/SP are very efficient in energy consumption (pJ/bit) but inefficient in area consumption. No local storage is distributed to the PE to minimize the off-chip memory bandwidth and maximize the storage capacity. Instead, the entire area is used for the SRAM access to increase the storage capacity, as shown in Fig. 3.13. In NLR data flow, nothing stays stationary inside the PE module. Hence, there will be increased traffic on the SRAM

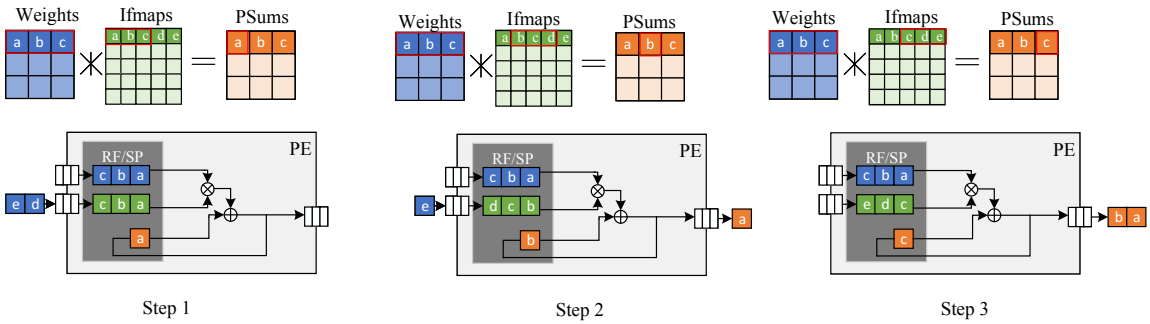


Figure 3.14: 1-D convolution reuse within PE for RS dataflow [28].

and the spatial array. All the activations are multi-casted in NLR dataflow, filter weights are single-casted, and the PSums are accumulated across the PE array.

In [87], the filter weights and input activations are read from SRAM, processed by MAC units with custom adder trees that process the accumulation results in a single clock cycle. In contrast, the resulting PSums or the output activations are stored in the global memory. Another example is DianNao [88], which reads the input activations and filter weights from the SRAM and processes them through the MAC units with custom-built adder trees. In addition, DianNao implemented specialized registers to store the PSums in the PE array, which further helped to reduce the energy consumption of accessing the PSums.

3.3.4 Row Stationary (RS) Dataflow

An RS dataflow maximizes the reuse and accumulation at the RF/SP level for all data types (weights, input activations, and PSums) to improve energy efficiency. Therefore, the RS dataflow differs from the WS and OS dataflow, as these dataflows optimize the weights and PSums, respectively.

For each PE processing a 1-D convolution, multiple PEs are combined to complete a 2-D convolution, as shown in Fig. 3.14. For example, three 1-D convolutions must create the first row of activations for a filter with three rows. Therefore, the PSums are vertically accumulated across three PEs to generate the first output row. To further develop the second row of output, we use another column of PEs, where the input activations from the three rows are shifted down by one row, and it uses the same row of filters to perform the three 1-D convolutions. Additionally, the other columns of PEs are added until all the outputs are computed.

To reduce the expensive memory access to the SRAM, the 2-D arrays of PEs also offer a different reuse. For example, each filter row is reused horizontally across the multiple PEs. Likewise, each row of the input activations is diagonally reused across the various PEs. In addition, each row of the PSums is accumulated across PE vertically. Hence, the 2-D convolutional data reuse and accumulation are reused to the maximum extent inside the 2-D array [89].

For each PE processing a 1-D convolution, multiple PEs are combined to complete a 2-D convolution, as shown in Fig. 3.15. To generate the first row of output activations with a 3-row filter, three 1-D convolutions are needed. To generate the first row of output activations with a 3-row filter, three 1-D convolutions are required. The PSums are vertically accumulated

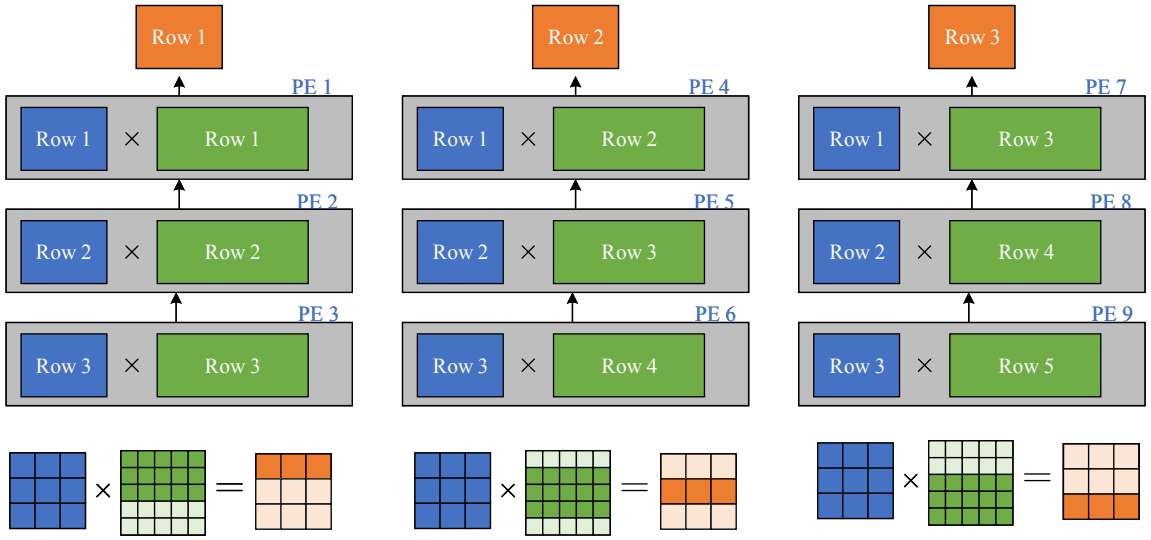


Figure 3.15: 2-D convolution reuse within PE for RS dataflow [28].

across three PEs to generate the first output row. To further develop the second row of output, we use another column of PEs, where the input activations from the three rows are shifted down by one row, and the same row of filters is used to perform the three 1-D convolutions. Additionally, the other columns of PEs are added until all the outputs are computed.

To reduce the expensive memory access to the SRAM, the 2-D arrays of PEs also offer a different reuse. For example, each filter row is reused horizontally across the multiple PEs. Each row of the input activations is diagonally reused across the various PEs. In addition, each row of the PSums is accumulated across PE vertically. Hence, the 2-D convolutional data reuse and accumulation are reused to the maximum extent inside the 2-D array [89].

As shown in Fig. 3.16, multiple rows are mapped into a single row to manage the high-dimensional convolution of CONV layers. As in the typical case of convolution computation, the 2-D convolutions are mapped into a group of PEs, and the extra dimensions are handled by concatenating the additional data. For the filter reuse technique, the different rows of the fmaps are concatenated through the same PE as in 1-D convolution. Similarly, for the Ifmaps reuse, different filter rows are interleaved and run through the same PE. Finally, to increase the local PSums accumulation inside the PE, filter rows and Ifmaps rows from different channels are interleaved and run through the same PE as 1-D convolution. The PSums from other channels are then accumulated inside the PE.

The total number of filters, Ifmaps, channels are dynamically processed simultaneously, and there is an efficient way of mapping these parameters to obtain the best energy efficiency. However, efficient mapping depends on the shape contour of the DNN under inference and the hardware resources. The hardware resources include the total number of PEs, the width, and the memory depth. Since most of the variables are not known beforehand, it is not possible to design a compiler, mapper, or control unit to take care of the mapping for the RS dataflow to improve the energy efficiency, as shown in Fig. 3.17.

One of the best examples of the RS dataflow is the multi-bit accelerator [30] [31]. The detailed explanation of the multi-bit accelerator is explained in Chapter 5. Hardware architecture must address two problems to support RS data flow. The first problem is to solve how the fixed-size

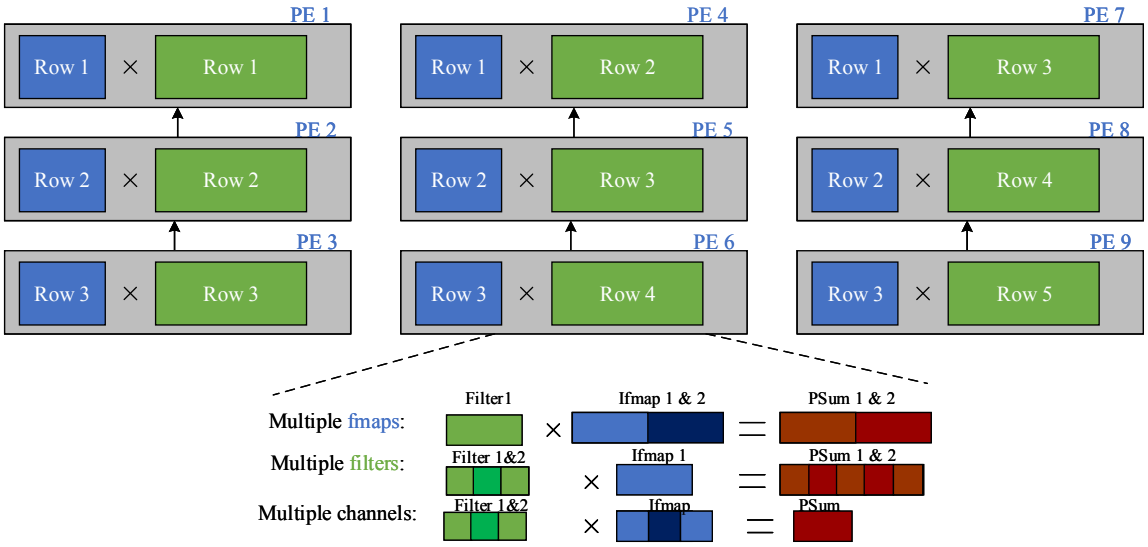


Figure 3.16: Multiple rows of different ifmaps, filters, and channels mapped into the same PE array in RS dataflow [28].

PE array can accommodate different layer shapes, and the second problem is to allow the data to be fed in a specific pattern by the DNN shapes. So, the main question is how a fixed design architecture would pass the data efficiently in different patterns.

The two mapping strategies are used to solve the first problem. In the first case, the replication strategy can be used to map the shapes that do not use the entire PE array. For example, in the third and fifth layers of the AlexNet, each of the 2-D convolutions uses the 13 × 3 PE array. For the accelerator with 16 × 16 PE array as in [30], the entire architecture would be replicated four times, running on different filters and channels for each application. The second strategy involves the folding technique, where, for example, in the second layer of the AlexNet, it requires a total of 27 × 5 PE array to complete the 2-D convolution. To fit the 16 × 16 PE array as in [30], it needs to be folded into two parts, i.e. 14 × 5 and 13 × 5, and each part is vertically mapped into the PE array. Since few PEs are not used, they can be clock-gated or connected to an Integrated Clock Gating (ICG) cell to avoid unnecessary dynamic power consumption. Both the folding and the replication strategies are equally beneficial. However, the folding technique is slightly better regarding power consumption, as shown in Fig. 3.18. Hence, this strategy is used extensively in the multi-bit accelerator, as shown in Chapter 4. The power consumption for different neural networks using row stationary dataflow is shown in Fig. 3.19. As shown in Fig. 3.10, Fig. 3.12 and Fig. 3.19, the RS data flow is the most efficient data flow both in terms of power and area consumption.

3.3.5 Comparison of different data flows

We compare the RS data flow with the other data flows in DRAM access and energy consumption for different layers of AlexNet with a PE size of 1024 and batch size of 1.

DRAM access: DRAM access has a more substantial impact on the energy efficiency since the energy needed for DRAM access is more than the on-chip data movement between the SRAM and the PE. The Table 3.2 shows average DRAM access per operation for the four different data flows. RS dataflow is 4.92 × better than WS dataflow, 0.88 × better than OS

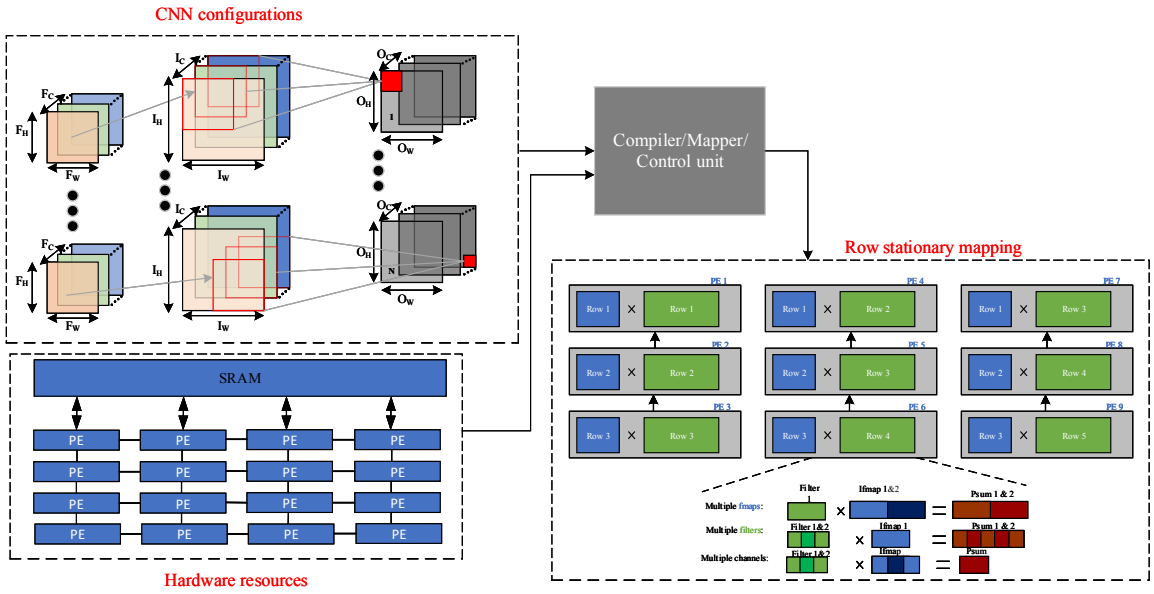


Figure 3.17: Mapping the CNN configuration with available DNN resources for row stationary dataflow [31] [30] [28].

Table 3.2: Average DRAM access per operation for different dataflow with AlexNet using the Virtex Ultrascale FPGA.

DRAM access for PE size 1024 with batch size 1 for AlexNet			
Dataflow	Memory Read	Memory Write	DRAM access/op
RS	0.001	0.007	0.008
WS	0.0025	0.16	0.1625
OS	0.003	0.006	0.009
NLR	0.0012	0.007	0.0082

dataflow, and 0.975 × better than the NLR dataflow. Using RS dataflow has less on-chip storage than others, highlighting the importance of co-designing the architecture with an efficient data flow.

Energy Consumption: Normalized energy consumption per operation is shown in the Fig. 3.20. From the figure, the RS data flow is 0.28 × - 0.52 × better than the other data flows due to the low-cost data movement.

3.4 Data Processing for DNNs

In this section, we further discuss the techniques to improve the energy efficiency of the DNN accelerator by bringing the DRAM closer to the computation unit or integrating the computation inside the memory. The latter is called *in memory computation*. Large-scale efforts have been made in most embedded system applications to bring the calculation inside the data source, such as the sensor unit, where the data is first collected. This section will discuss computing and data closer to lowering the data movement inside the processor using mixed-signal and advanced memory techniques.

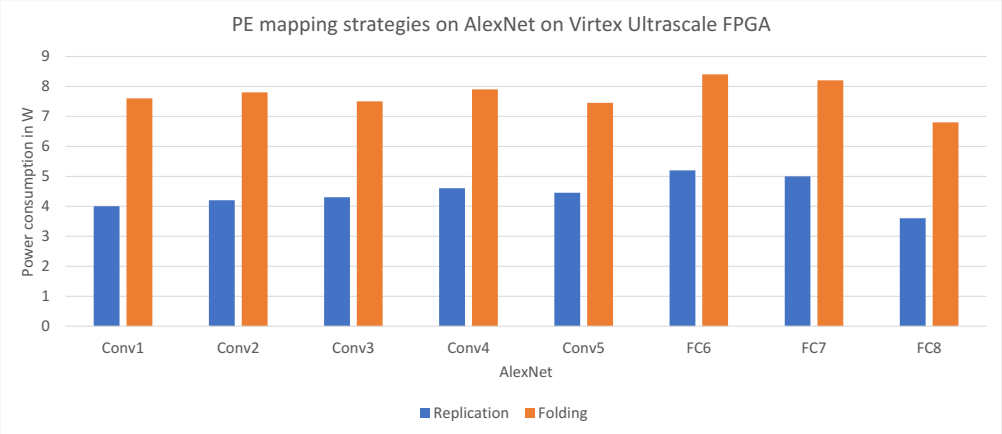


Figure 3.18: PE mapping strategy for AlexNet on the Virtex Ultrascale FPGA.

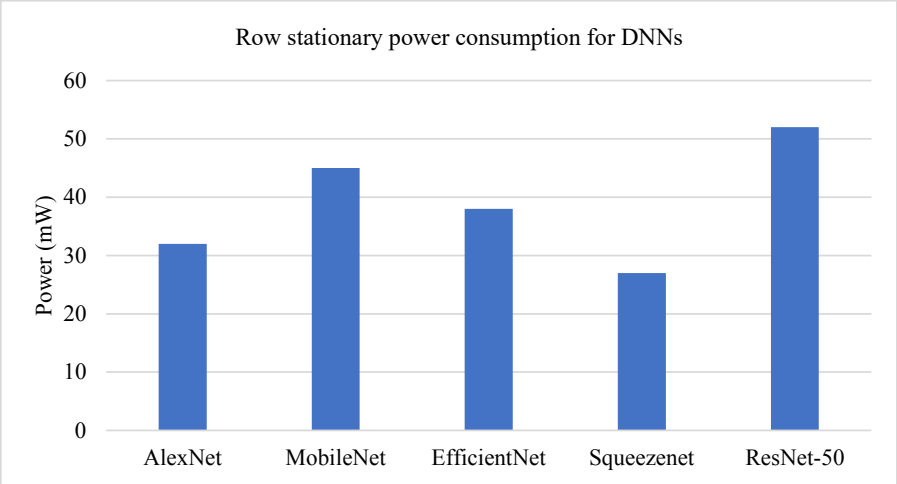


Figure 3.19: Power consumption for different neural networks using row stationary data flow.

Most of the previous works use analog processing techniques, which leads to increased sensitivity and non-idealities [27] [29] [89] [90]. Effectively, the computation is performed at a lower precision, which we will consider during the training of the DNNs [29] [89]. By performing at lower precision, the total amount of memory bandwidth is reduced, increasing the power efficiency for neural networks for reduced performance computation. Both neural network training and inference applications can effectively utilize memory bandwidth by packing more numbers into each byte than a complete precision computation. As a result, different neurons are implemented, increasing the total cost of operation [91]. Also, DNNs are trained in the digital domain; thus, analog processing would lead to additional operational costs for analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC).

3.4.1 DRAM

Advanced memory techniques can help to reduce the energy access for high-density memories such as DRAMs. The embedded DRAM (eDRAM) offers a high-density memory on-chip to

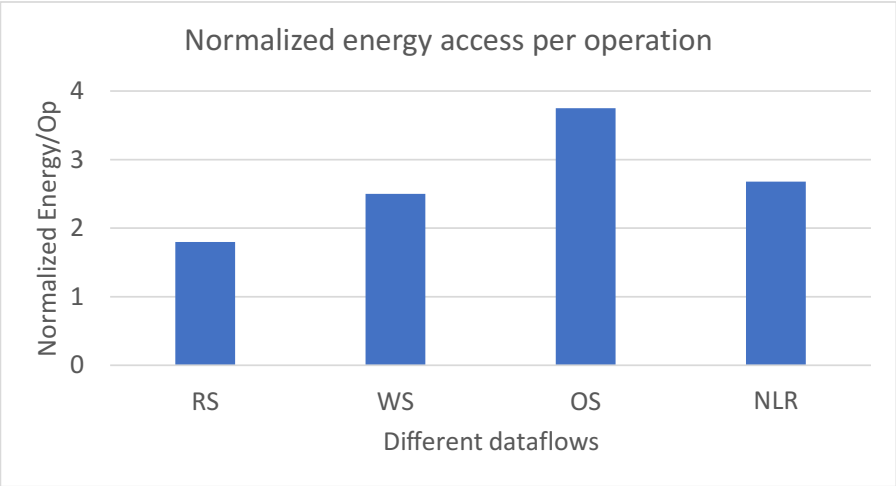


Figure 3.20: Normalized energy access for PE with an array size of 1024 and batch size of 1 in Virtex Ultrascale FPGA for different dataflows.

avoid high switching capacitance [92]. The eDRAM is 2.85× higher density than SRAM and 322× more energy efficient than DDR3 DRAM [90]. Also, eDRAM offers high bandwidth and lower latency compared to DRAM. For the DNN processing, eDRAM stores 10MB of weights and activations on-chip to avoid unnecessary off-chip access, as shown in DaDianNao [88]. The main drawback of the eDRAM is that it possesses a lower density than the off-chip DRAM and can increase the overall cost of the chip.

3.4.2 SRAM

SRAM is the most common on-chip storage platform to store the Ifmaps, weights and PSums. The storage depends on the neural network under inference and the hardware computation unit. Most of the SRAM are generated from the memory compiler provided by the foundry. The available size depends on the width and the depth of the memory. It occupies a significant amount of silicon area. In addition, it contributes significantly to the power consumption along with the computation unit, as shown in Table 4.21.

3.5 Energy Efficient Co-Design of DNN Models and Hardware Architecture

DNNs undergo training to maximize accuracy without considering much of the hardware complexity. However, since data movement and computation are complicated, these models can be challenging to implement and deploy. Therefore, it is imperative to co-design the DNN models and the hardware to maximize the accuracy and throughput while reducing the energy and silicon area. Some of the co-design approaches are grouped into the following categories:

- Reduced precision of operations and operands. This includes switching from the floating-point to the fixed-point approach, reducing the bit width, non-linear quantization, and weight sharing.

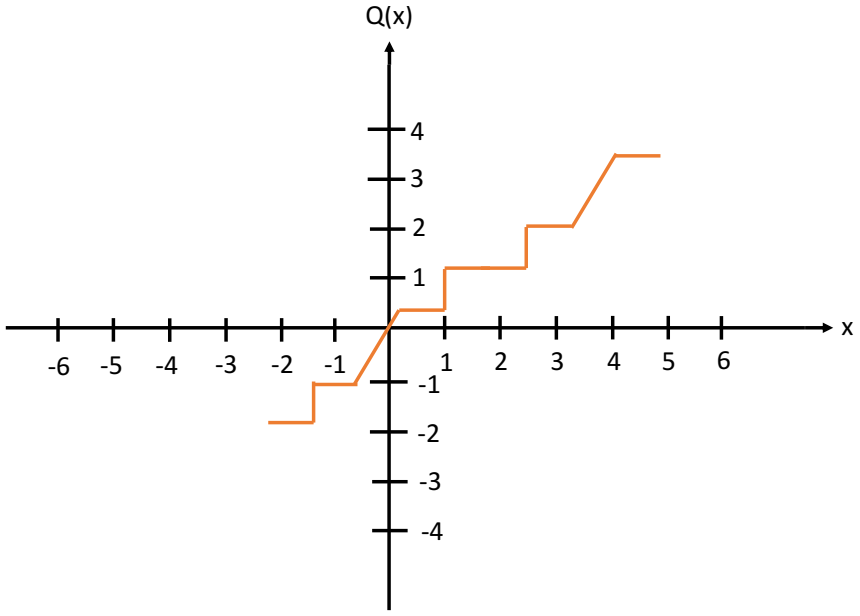


Figure 3.21: Log quantization.

- A total number of reduced operations and size for the DNN model size. This includes techniques such as compression, pruning, and squeezed network architecture.

3.5.1 Reduced Precision

Quantization involves the mapping of data into a smaller set of quantization levels. The main goal is to reduce the error between the original and the reconstructed data from the quantization levels. The actual number of quantization levels contemplates the precision and the number of bits required to represent the data (usually log_2 of the total number of levels). Thus, the reduced precision mainly refers to the reduced number of levels and bits. Reducing the number of bits gives benefits such as reduced storage and computation requirements.

There are diverse ways to map the quantization levels. One of the simplest methods is the linear mapping of each quantization level with a uniform linear quantization distance. Therefore, the quantization levels are spaced uniformly; however, it has more to do with the quantization error. A second method involves using a mapping function, such as the log function, where the distance varies by a small margin. Simple logic operations like shift operations are often used to implement this kind of mapping, which is advantageous. However, one drawback is that log quantization places more boundaries for low-magnitude values than for high-magnitude values. Often, this results in lower quantization error when the error values are low. However, inappropriate errors when the error values are high can undermine the accuracy of log quantization to a large extent, leading to worse performance than linear quantization [93]. A substitute method involves a more complex mapping function that can be used when the quantization levels are determined from the data, using mainly the k-means of clustering. This method uses a look-up table approach [54] [94]. Fig. 3.21 and Fig. 3.22 show the pictorial representation of log and linear quantization.

Quantization can also be fixed, i.e. the same quantization method is used for different types of data, layers, filters, and channels in the network, or it can be variable, i.e. different

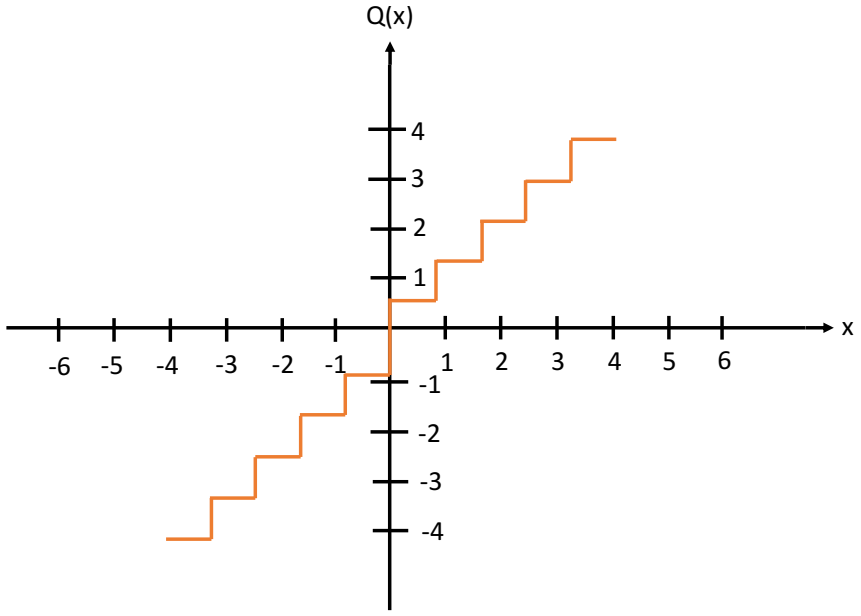


Figure 3.22: Linear quantization.

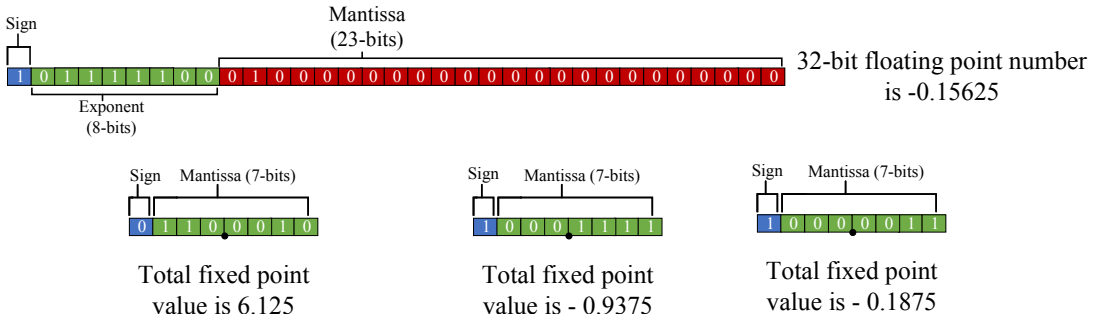


Figure 3.23: Fixed and floating point operations.

quantization methods can be used for weights, activations, and different layers, filters, and channels in the network [29].

Reduced precision experiments mainly focused on reducing the precision of the weights rather than that of the activations. Since weights increase storage capacity, activations effect on storage capacity depends on the data flow and the network architecture. Due to gradients' sensitivity to quantization, reduced precision also aims to reduce the precision of inference and training.

3.5.1.1 Linear Quantization

The first step in reducing the precision is to convert values from the floating-point to fixed-point. A simple example of floating and fixed-point numbers is shown in Fig. 3.23. In floating point numbers, the sign bit is $(-1)^s$ and the exponent is $(e-127)$. The mantissa is m and covers the range of 10^{-38} to 10^{38} [29] [95] [96].

Similarly, an N-bit fixed-point number is represented as $(-1)^s \times m \times 2^{-f}$, where f determines the location of a decimal point used as a scaling factor. For example, in an 8-bit integer, when

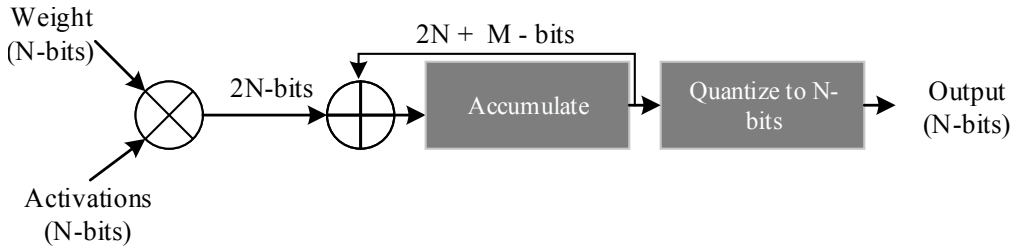


Figure 3.24: Reduced precision for MAC units [29].

$f = 0$, the dynamic range of the fixed-point number is -128 to 127, similarly when $f = 10$, then the dynamic range is -0.125 to 0.12402348 [29] [62] [97]. The dynamic fixed point allows the value of f to be dynamically changed for the weights and activations. It allows the dynamic range to be varied from layer to layer. As per [98], the bit width can be reduced to 8 bits for weights and 10 bits for the activations without fine-tuning with both weights, and activations can reach up to 8 bits [99].

A reduced bit width like an 8-bit fixed point adder consumes approximately 3.8× lower area and 3.3× lower energy than a 32-bit fixed-point add. Similarly, it consumes 30× lower energy and 116× lower area than the 32-bit floating-point add [26]. The area and energy numbers scale linearly with the number of bits for fixed-point add.

An 8-bit fixed-point multiplier consumes 15.5× lower energy and 12.4× lower area than a 32-bit fixed-point multiplier. Similarly, it consumes 18.5× lower energy and 27.5× lower area than a 32-bit floating-point multiply. The area and energy numbers scale quadratically with the number of bits for fixed-point multiply [28] [31] [90] [100].

Additionally, reducing precision reduces the memory's area and energy cost since energy and area scale approximately linearly with the number of bits. However, the most important thing is that changing the bit-width from a floating point to a fixed point is the key to reducing the energy or cost of the memory.

In the MAC operation, the resultant product bits of the multiplication are typically higher than the activations and weights. To ensure there is no precision loss, weights and the input activations with N-bit fixed point would require N-bits × N-bits multiplication to generate a 2N-bit output result. The output would then be accumulated with 2N+M-bit precision. The largest filter size calculates the value of M [31]. After the accumulation, the precision of the final output activation would be in the order of N-bits, as shown in Fig. 3.24. The reduced output precision does not significantly impact the accuracy, only if the distribution of the weights and activations are centered around zero. The accumulation would not move in one direction, which is particularly important when batch normalization is used.

The reduced precision is used in commercial platforms for DNN processing, like Google's TPU unit designed for 8-bit fixed-point arithmetic. Similarly, Nvidia's PASCAL, developed in April 2016, also performs the inference on an 8-bit integer. However, in general, most general-purpose platforms like CPUs and GPUs employ 8-bit computation to increase the overall throughput, as four 8-bit operations are more beneficial than one 32-bit operation for one clock cycle.

The precision can be further reduced by employing a single bit known as binary nets. The weights are quantized only to +1 and -1 values in binary nets. Multiplication in the MAC is reduced to addition and subtraction with binary weights. The binary nets provide an accuracy loss of 29.8%, much more than the CNN because the weight values are limited between +1, 0, and -1 [101]. The usage of binary nets is explored in a few publications like YodaNN [102], which uses binary weights. Similarly, BRein [103] uses both the binary weights and activations.

3.5.1.2 Non-Linear Quantization

In the previously published works that involved linear quantization, all the levels were uniformly distributed. In a few of the research works like [54] [104], the distribution of the weights and activations are not uniform. Hence, a non-linear quantization improves the accuracy. The two most common approaches are:

1. **Log Domain Quantization**: If the quantization levels are based on logarithmic distribution, then the weights and activations are distributed equally across various levels, and each level is more efficient with lower quantization error. For example, using 4-bit linear quantization results in 27.8% loss in accuracy versus 7% loss in logarithmic quantization for VGG-16 [94] because logarithmic quantization enables the network to achieve higher classification accuracies than low fixed-point resolutions. Also, logarithmic quantization helps reduce bulky multipliers. In addition, when weights are quantized in the powers of two, the multiplication is replaced with a bit-shifting operation [99] [104].

2. **Weight Sharing**: Weight sharing enables several weights to share a single value. This reduces the number of unique weights in a particular layer or weight. The weights can be grouped using a k-means clustering algorithm or hashing algorithm. In unsupervised machine learning, the k-means algorithm groups the unlabeled data into different clusters. It is used to solve the clustering problem in machine learning by enhancing the compression techniques. More details are found in [54].

   Similarly, the hashing algorithm assists in vectorizing features by turning arbitrary features into indices in a vector [105]. The shared weight and the index indicate the weights for each filter position. The weights are stored in the weight-sharing format. This is performed in two different steps to fetch the weights: First, read the weight index, and second, read the weight index and the shared weights. This technique will help reduce the cost of reading and storing the weights if the weight index is lower than the bit-width of the weight. The weight sharing does not reduce the precision of the MAC computation, but it would reduce the weight storage requirement.

3.6 Metrics for DNN Training and Inference

As significant research has been conducted on developing efficient DNN processing, it is imperative to consider specific metrics to compare the strengths and weaknesses of the different architectures and design techniques. Some metrics are accuracy, power consumption, throughput, latency, and area. The metric comparison is performed for both the DNN models and the hardware.

Table 3.3: Metrics for DNN models with AlexNet as an example.

Metrics	AlexNet	
	Dense	Sparse
Top-5 error	19.6	20.4
Number of CONV layers	5	
Depth of CONV layers	5	
CONV Filter Sizes	3,5,11	
Number of CONV Channels	3-256	
Number of CONV filters	96-384	
Stride	1,4	
NZ CONV weights	2.3M	351K
NZ CONV MACs	395M	56.4M
FC Layers	3	
FC Filter Sizes	1,6	
Number of FC Channels	256-4096	
Number of FC filters	1000-4096	
NZ FC weights	58.6M	5.4M
NZ FC MACs	14.5M	1.9M
Total NZ weights	61M	5.7M
Total NZ MACs	410M	58.3M

3.6.1 Metrics for DNN Models

The following metrics must be considered when evaluating the DNN model.

- The *accuracy* of the DNN models is characterized in terms of top-5 error on datasets such as ImageNet.
- The *network architecture* of the DNN models is reported based on the number of layers, filter sizes, the total number of filters, and the number of channels. The key parameters to determine each architecture's efficiency are accuracy loss, top-1 and top-5 error rates, and the total number of MACs.
- The *number of weights* has an impact on the storage requirement of the DNN models. Also, it is necessary to report the non-zero weights as they reflect the theoretical minimum storage requirement.
- The *number of MACs* is a critical metric to be reported as it indicates the total number of operations and the throughput. Also, it is necessary to report the non-zero MACs as they reflect the minimum compute requirements.

An example is shown in Table 3.3 for AlexNet for the various metrics. The accuracy is reported in terms of top-5 error, the number of weights, and the MACs are coherent. The non-zero (NZ) operations reduce the total number of MACs and weights. The total number of NZ MACs depends on the input data.

3.6.2 Metrics for DNN Hardware

DNN hardware efficiency is evaluated by considering the following metrics:

- The *power and energy consumption* are the most critical metric for the DNN hardware. Here, the DNN model specifications will be provided, including the layers and the bit precision information supported by the hardware during the measurement. Also, the DRAM access should be included in the power consumption as they occupy a significant amount of system power. The power consumption metric mainly reports the power consumption information of the chip and the total memory access between the off-chip and on-chip memories.
- The *latency and throughput* metric are reported among several DNN models in terms of batch size and actual run time, which mainly accounts for mapping and memory bandwidth effects. This is a critical metric compared to peak throughput.
- The *implementation cost* of the chip mainly depends on the area efficiency, which accounts for both the memories (i.e. registers and SRAM) and the amount of computation logic. Therefore, it should be reported in terms of core area or total area of the chip in squared millimeters.

In terms of the cost, different platforms have different implementation standards. For example, for FPGA implementation, the specific device should be reported along with resource utilization like BRAMs, FFs, LUTs, and DSPs as explained in Chapter 4 and 5.

In Tables 4.19 and 5.5, each processor should report various specifications for each metric. Including all metrics and specifications is crucial for evaluating design trade-offs since they reveal how well the inference architecture is designed regarding resource usage, the area occupied, latency, and throughput. It is essential to report the accuracy of the inference, as it shows how efficient processors work with power and area consumption. Without accurate information, one can quickly run a simple DNN and claim low power, high throughput, and small form factor processor. As a result, one needs to consider the off-chip memory bandwidth since one can build a processor with low cost, high throughput, and low power with few multipliers. However, when evaluating the system power, it is imperative to consider the off-chip memory access.

In summary, the evaluation process of the DNN system should be of the following order:

1. The level of accuracy the system can provide during the inference.
2. The latency and throughput on how fast the system can infer the real-time tasks.
3. The device's form factor will be dictated primarily by its power and energy consumption.
4. The total cost primarily dictated by the chip area determines how much one should pay for the required solutions.

3.7 Summary

Data flow forms an integral part of the DNN accelerators. It mainly dictates the overall performance and the energy efficiency of the system. It is one of the critical attributes of the

DNN processor and is analogous to the general-purpose processors. Based on the following insights, we have reported the four different dataflow techniques and provided each dataflow's equivalent dynamic power consumption. Among the four dataflows, the row stationary dataflow is very efficient compared to other dataflows. They are 30% better than the weight stationary dataflow and 38% better than the output stationary dataflow. In addition, this chapter explains the different quantization techniques that are important during the inference, and how these techniques influence the truncation is explained in the next chapter. Finally, this chapter offers a complete insight into the metrics like accuracy, throughput, and power consumption that determine the processor's efficiency.

4

A Power Efficient Multi-Bit Accelerator for Memory Prohibitive Deep Neural Networks

4.1 Introduction and Motivation

Deep neural networks (DNN) have been an essential part of computer vision technologies due to their high accuracy in image classification. Today, deep neural networks have been found in applications such as face recognition in mobiles, natural language processing, speech-to-text conversion, user identification, surveillance, military applications, and entertainment systems. Currently, research is conducted in the DNN thanks to its robustness in extracting vital information from the data. In addition, the brain-inspired techniques of deep learning algorithms help tackle object recognition and image classification problems. The most commonly used CNN networks for image classifications with high accuracy are AlexNet [17], VGG-16 [106], YoloNet [107], GoogleNet [25], MobileNet [73], SqueezeNet [52] and EfficientNet [76].

Since the advent of back-propagation in neural networks by Hinton et al. [109], a significant amount of research has been taking place both in academia and the industry. Two decades ago, LeCun et al. [110] performed the classification of handwritten digits with less than 1M parameters using LeNet-5 architecture. In 2006, Hinton et al. [111] stacked multiple RBM(Restricted Boltzmann Machine) layers for the efficient training of a large amount of data and termed them as Deep Belief Nets(DBN). Similarly 2011, Glorot et al. [112] implemented the ReLU activation function to solve the vanishing gradient problem. In ILSVRC-2012 [113], Krizhevsky et al. [17] introduced AlexNet with 60M parameters trained for 1000 different classes. Recently, in ISLVRC-2016, PSPNNet [114] with 273M parameters won the challenge. In 2014, Goodfellow et al. [115] introduced a Generative Adversarial Network (GAN) for generating real data from random variables.

Table 4.1: Different network characteristics with data type of 4 bytes [30] [31] ©2021 IEEE.

Network	Layers	Weights (MB)	Total Parameters
AlexNet [17]	8	250	60 Million
VGG-16 [106]	8	594	138 Million
GoogleNet [25]	15	51.3	12.8 Million
YoloNet [107]	26	206	51.5 Million
ZFNet [108]	16	500	14 Million

Even though these models are robust and accurate, they are computationally expensive when implemented on-chip, as shown in Table 4.1. These networks consume a significant amount of memory in weights and input activations. As a result, they have to be stored in the off-chip DDR3 DRAM memory. An obvious outcome is increased energy consumption while accessing them from the off-chip. Table 4.1 lists the attributes of the most commonly used CNN for image classification. These weights indicate the maximum size required to store the weights in DDR3 memory, and the network parameters are computed on all layers.

The present state of art CNNs require hundreds of megabytes of weight to perform billions of operations on the FPGA. This leads to a significant bandwidth and bottleneck between the on-chip SRAM and off-chip DRAM. As a result, the total energy for the data movement is 10× greater than the computation as shown in Table 3.1 and as explained in [26]. Therefore, designing a scheme that can effectively reduce the data energy requirements between on-chip and off-chip memories is vital to address these hardware design challenges. The energy cost of the data movement can be reduced in power-sensitive applications by systematically reusing data in a multilevel memory hierarchy. Furthermore, we can exploit the parallelism in the processing element (PE) architecture to improve energy efficiency. In addition, we use adaptive data processing and bit-width reduction as we move layer by layer in the neural network.

Previous works have employed specialized hardware to accelerate the DNN models [116] [117]. The main emphasis of these architectures was on accelerating the dense models, hindering their utility due to the high energy cost of the external DRAM. As a result, it is more important to reduce the energy and power costs of these powerful DNN models is more important. However, the forward propagation of CNNs has very high computational complexity and requires adequate real-time performance. Moreover, when considering applications with large image dimensions like 227×227 sized pixel RGB image, the number of arithmetic operations such as MAC exponentially increases. Hence, the proposed [87] [90] [118] classical general purpose (GP) processors, FPGA accelerators, and ASIC accelerators do not meet the requirement of the real-time applications that are very sensitive regarding power usage. Therefore, we extend the multi-bit architecture in our previous work [30] with more details on truncation analysis. The PE architecture was also modified for lower power consumption and increased throughput by having larger PE arrays and improved pipelined dataflow movement for enhanced performance and lower power consumption. The main limitations of the previous works are:

1. **Lower precision scalability:** Many FPGA accelerator architectures support fixed bit precision like 4, 8, and 16 bits. However, for two reasons, most deep neural network architectures will not have the required weight precision. The first reason is that different

DNNs have different optimal weight accuracy, [119] [120], and overall different accuracy for different bit widths. Secondly, every layer in a DNN has a different accuracy loss [121]. For example, AlexNet has a 10% accuracy loss for 4-bit weight precision compared to 32-bits. Similarly, ResNet has 8%, GoogleNet has 7%, and LSTM has 12% accuracy loss for 4-bit weight precision compared to 32-bits. In comparing 4-bit and 32-bit accuracy, the average accuracy loss for all layers is considered. The lower bit-width weights improve the energy efficiency of the DNN accelerator due to the reduced data movement. The desired weight precision also depends on the target accuracy of a particular DNN. So, it is important to design an accelerator that supports variable bit-width weight precision to support optimal energy-accuracy trade-off. However, [86] [100] [118] did not design the architecture to support the optimal energy-accuracy trade-off.

2. **Different convolution and FC architecture:** The caffeine processor [122] had a separate accelerator design for convolution layers (CLs) and FC layers (FCLs), as CLs are computationally dominant and FCLs are memory dominant. The PEs used for CLs are not re-configurable for FCLs and vice versa. This reduced flexibility leads to deficient performance.

In this chapter, the limitations of the previous works are overcome by two distinct features. First, we propose a multi-bit architecture employing different bit widths for each layer without compromising the overall performance. Secondly, the architecture supports CLs and FCLs for the different bit widths. Prior architectures primarily concentrate on fixed bit width processors, improving the data transfer bandwidth and reducing the computation time in the processing element part. The objective of these processors was mainly to capitalize on the data flow movement, to improve the total number of operations per second per watt [27], and to eliminate unwanted computations. However, networks like RNN and LSTM widely operate on FC layers. This requires a significant amount of data movement between the off-chip DRAM and the on-chip SRAM as the data reuse is impossible. So, in the fixed-bit architecture, every clock cycle, a MAC operation increases the power consumption. To overcome this drawback, the FC layers have a lower bit width, which reduces energy and resource consumption without compromising accuracy. In our architecture, the inference starts with a larger bit width in the convolution layers while successfully reducing the bit width of the following FC layers. This technique saves 50% of the total power consumption and 60% of the resource utilization on the FPGA [31] [123].

In the following section, we describe the entire multi-bit architecture and the ALU operations of each PE. Section 4.3 discusses the comparison metrics used for evaluation, followed by the methodology. Section 4.4 shows the results for different bit combinations, ASIC results, and the comparison with other SoA designs. In Section 4.5, we present our conclusion. All the content of this chapter is published in the journal article [30] [31], as the author of this thesis is also the first author of the published journal.

4.2 Multi-bit Architecture

The top-level architecture of the multi-bit accelerator with memory hierarchy is shown in Fig. 4.1. It has two clock domains: a clock domain for the processing stage (operating at

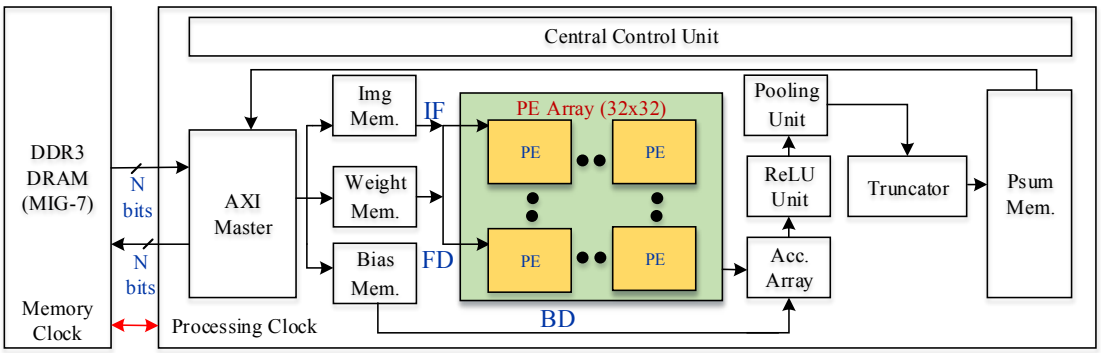


Figure 4.1: Multi-bit accelerator architecture [30] [31] ©2021 IEEE.

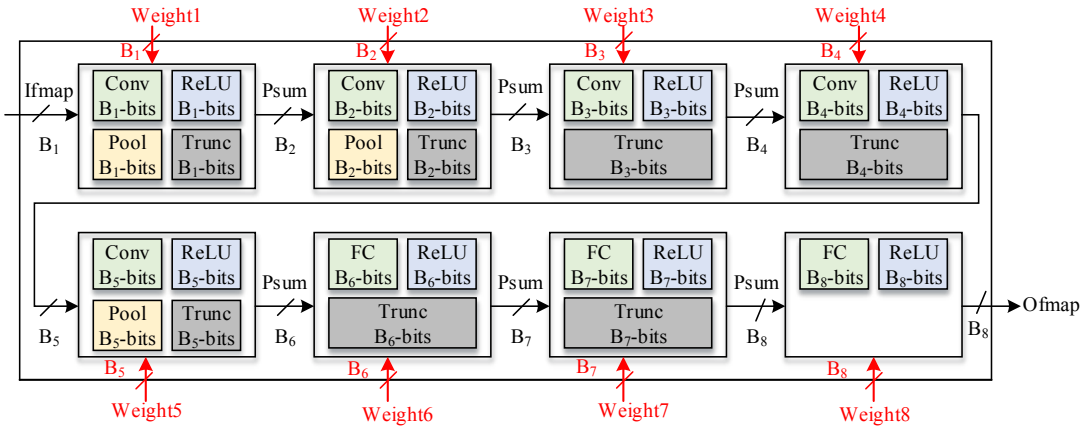


Figure 4.2: Data flow model for the multi-bit accelerator architecture for AlexNet [17] [30] [31] ©2021 IEEE.

180MHz for FPGA implementation and 800MHz for ASIC implementation) and a memory clock domain for the communication with MIG-7 (Memory Interface Generator) interface to access the DDR3 DRAM operating at 200MHz. The two domains communicate with each other using the AXI clock converter. The processing clock domain mainly works on the PE array of 1024 PEs arranged in a 32×32 square pattern while handling data movement between the memories and other system modules. A 375KB of on-chip SRAM is used to store the Ifmaps data, filter data (FD), and the bias data (BD) in a SRAM of each 125KB. An accumulator array module adds the convoluted data with the BD. ReLU module and pooling module are used to perform the non-linear activation and pooling functions. The truncator is used to truncate the PSums results, and PSums memory of 125KB is used to store the PSums, and final Ofmaps is fed back to the off-chip DRAM. Finally, the AXI master communicates with the external DRAM memory (through the MicroBlaze and MIG-7) and the architecture. The corresponding data flow model of the architecture for AlexNet DNN is shown in Fig. 4.2. The Ifmap data comprising B_1 bits is inferred with weights of B_1 bits in the first CONV layer of the AlexNet to generate a truncated PSums results of B_2 bits. These PSums results are fed to the off-chip DRAM as these results are large and cannot fit on a 125KB SRAM. For the inference of the next layer, these PSums results of B_2 bits are inferred with the weights of

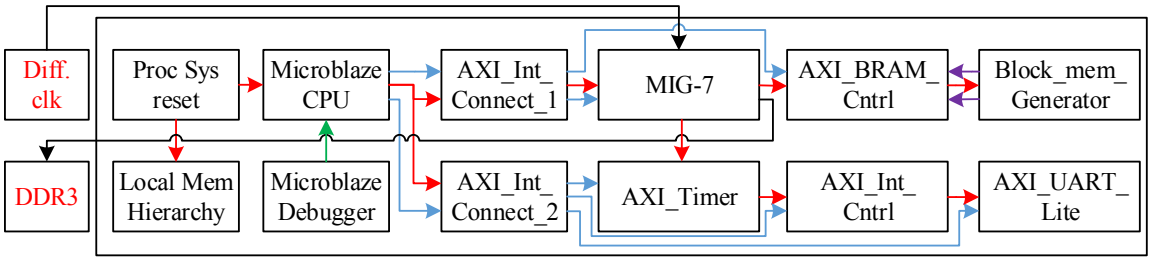


Figure 4.3: AXI communication protocol with DDR3 using MicroBlaze CPU [31] ©2021 IEEE.

B_2 bits to generate a new PSums of B_3 bits. This cycle continues for all eight layers of the AlexNet. No additional training was performed for reduced bit width on any DNN models, as the weights/biases were truncated off-chip from the 32-bit reference weights. The multi-bit architecture supports DNN models like AlexNet, EfficientNet-pointwise, SqueezeNet, and MobileNet-pointwise. Our architecture can also help the depthwise operation by computing each channel of the DNN model separately without accumulating them at the end. However, the main focus is improving the movement of data flow and reducing power and resource consumption. The depthwise operation adds to the overhead of extra resources and power not supported in this architecture.

4.2.1 DDR3 DRAM and MIG-7 Memory Controller

DDR3 DRAM is a double data rate architecture that supports high-speed operation up to 200MHz. It performs a single read/write operation on a single 8-bit-wide data bus at a core voltage of 1.5V. Each 8-bit wide data is transmitted and received with a data strobe signal (DQS). The DQS is a differential bi-directional signal, where the edge is aligned with the read clock and the input with the write clock. The DQS uses both the clock edges [124]. The DQS has mainly two essential features:

1. During the write operation: The FPGA controller drives the signal and will center aligned with the DDR data.

2. During the read operation: The signal is driven by the DDR3 SDRAM chips, and it is edge-aligned with the change in the DDR data.

The DDR3 DRAM operates on a differential clock, and the read and write access is burst-oriented. In addition, it is a pipelined, multi-bank architecture that performs the concurrent operation. Thereby providing high bandwidth and a low pre-charge and activation time. Furthermore, a self-refresh mode is provided with the power-saving and power-down modes.

DDR3 memory is furnished through MIG-7. The MIG-7 memory controller uses the AXI4 interface for communication. By nature, the AXI4 interconnect block in our architecture follows a point-point protocol. Therefore, the AXI4 user interface signals communicate through the AXI4 interconnect block and the memory controller to access the data from the DRAM. The complete architecture of AXI4 communication protocol with DDR3 using MicroBlaze CPU is shown in Fig. 4.3, with differential clock signal operating at 200MHz. In Fig. 4.3, the red line indicates the reset signal from the processing system reset unit. The green line indicates the debugging signal from the MicroBlaze debugger and the CPU. The blue line indicates the

master the slave communication protocol, and the purple line indicates the communication between the block memory generator and the AXI BRAM controller.

Microblaze CPU is a 32-bit embedded RISC processor core used for the optimized implementation in the FPGA. It enables the communication between the AXI user interface and the off-chip DDR3 DRAM through the MIG memory controller. The main objective behind using this processor is the advanced flexibility in providing new instructions every clock cycle and maintaining the single-cycle throughput. The processor's local memory bus (LMB) provides fast on-chip storage. In addition, the CPU offers AXI protocol to communicate with the user interface signals. The MicroBlaze implementation in our architecture consumes 19 BRAM and 5 Digital Signal Processor (DSP) resource elements.

The two AXI4 interconnects are mainly used for accessing memory through the cache and accessing peripherals in the architecture. The AXI_Interconnect_1 bridges the MicroBlaze processor instruction and data caches to write the data into the AXI4 BRAM module and the off-chip DDR3. The interconnect uses the instruction cache and data cache buses to write the data from the master memory (user interface signals into AXI BRAM) into the slave memory (off-chip DDR3). AXI_Interconnect_2 connects AXI_timer, AXI_UART_Lite, and AXI_Interrupt_Controller to the AXI peripheral bus of the MicroBlaze processor.

MIG-7 is implemented to generate DDR3 DRAM in the system and uses a memory type of SODIMMs. MIG consists of an AXI4 slave interface to communicate with the DDR3 devices, and the data width is 32 bits. The clock frequency of the controller is 200MHz, and it is mainly differential. Once re-configured as DDR3 DRAM, the MIG controller drives the clock in the system and acts as a slave of the MicroBlaze processor's instruction cache and data cache.

The remaining modules: *Microblaze debugger* enables the JTAG-based debugging on the MicroBlaze processors. *Processing system reset* is a soft IP core in Xilinx FPGA that handles the reset conditions for any given system. The IP can handle multiple input conditions and generates appropriate resets at the output. The *local memory hierarchy* consists of the local memory bus and its respective BRAM controller. The primary function of the *AXI_BRAM_Controller* is as an AXI endpoint slave to facilitate the integration between the AXI interconnect and the local memory hierarchy. AXI_BRAM_Controller supports a memory size of up to 8MB and performs single and multiple burst transactions. *AXI_Interrupt_Controller* acknowledges the interrupts generated from the AXI_Interconnect. The core of *AXI_UART_Lite* provides a controller interface between UART signals and the AXI interface and also provides the feature of a controller interface for an asynchronous data transfer. It supports a full-duplex communication protocol with a baud rate of 9600. Finally, *AXI_timer* IP has a timer module associated with the load register that is used to hold the counter's initial value for the event generation. Also, it captures the value depending on the operating mode of the timer. Each module's resource and power usage is mentioned in section 4.4.1.

4.2.2 Central Control Unit

Our proposed accelerator has two levels of control hierarchy through the central control unit. The first level controls the movement of data traffic between the off-chip DDR3 to the on-chip SRAM through the AXI master. At the second level, it 1) performs the First In First Out

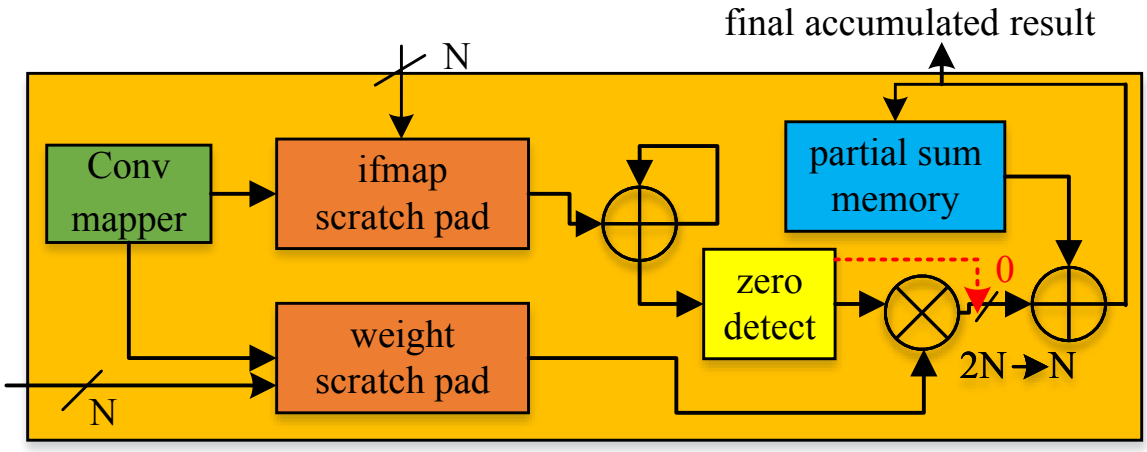


Figure 4.4: PE architecture [30] [31] ©2021 IEEE.

(FIFO) operation of loading the Ifmaps, weight data, and PSums into the individual memories. Since each memory is only 125KB, entire activations, weights, or the PSums cannot be directly loaded into the memory. As a result, the control unit controls the loading of the data into the memory using a synchronous FIFO. The FIFO determines whether the individual memory is "full" or "empty," based on which the data is loaded from the off-chip DRAM. Similarly, based on the CONV and FC principle, each DNN model "stride" is taken into consideration and loaded into individual scratch pads of the PE, 2) maps the movement of Ifmaps and filter data through the individual PEs for convolution and FC operation, 3) ReLU and pooling mode of operation based on the layers of the neural network under inference, 4) performs truncation operation on the ReLU or the pooled data 5) writes the truncated data onto the PSums memory and moves the final Ofmaps onto the off-chip DRAM and 6) finally performs the dynamic reconfiguration of the bit width of the PE array and other architecture components for the next layer of inference. However, the PE array and the memory depth will be of fixed configuration, and only the bit width will be dynamically changed for the next inference layer.

4.2.3 Processing Element Array

The backbone of the multi-bit architecture is the processing element (PE) array that performs the computation of the CONV and the FC layers. The block overview of the PE architecture is shown in Fig. 4.4. The CONV mapper statically configures the regulation of the PE by mapping the FD by the sliding window mechanism for a particular convolution and FC layer. For example, in AlexNet, the CONV1 has a filter size of 11×11 with a stride of 4, so based on the stride value, the filter data is loaded into each PE, and the data is reused to minimize the energy requirement. The Ifmaps scratchpad is implemented in a 4KB SRAM, and the filter scratchpad is implemented in a 32b×11 register. The two scratch pads are separated to provide enough bandwidth for data movement. The sizes of the scratchpad described here are maximum as the bit width of the succeeding layers will be truncated, and the requirement will be lower than this.

The row stationary (RS) data flow method is employed in the proposed architecture. This technique provides the advantage of fixed reconfiguration into varied sizes and shapes for any

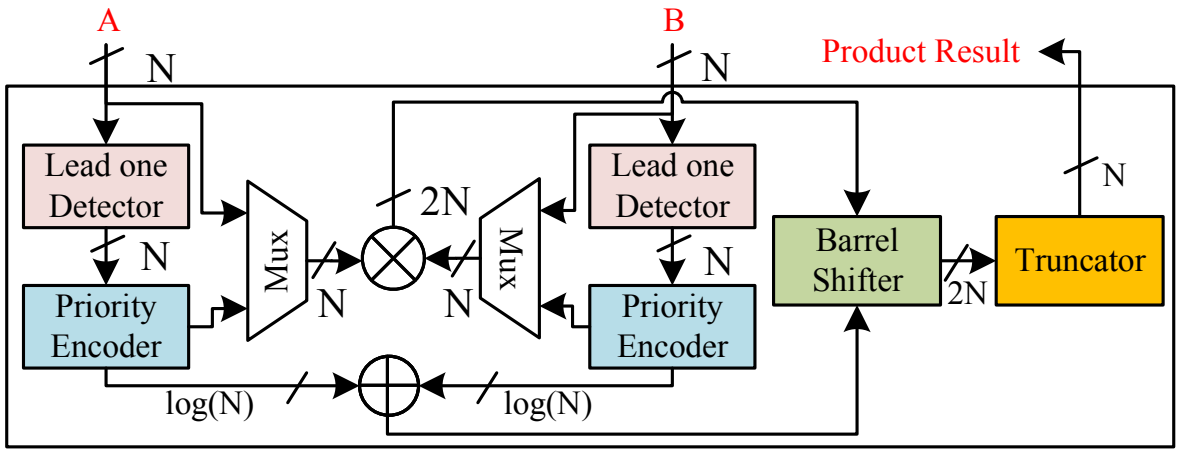


Figure 4.5: Approximate multiplier [125] [31] ©2021 IEEE.

given neural network layer under inference. The fixed reconfiguration involves the movement of input activations, weight data, and the PSums results to and from the DRAM. The central control unit maps the entire reconfiguration operation. Furthermore, the RS dataflow provides the best energy efficiency as it reduces the data movement among the Ifmaps, FD, and the Ofmaps, as all the FD and the Ifmaps are stored in the scratchpad in each of the individual PEs.

The other dataflows, like weight and output stationary, were implemented on the MobileNet and other DNN models. Based on this, the corresponding dynamic power consumption was measured for each data flow. The weight stationary dataflow, from the SRAM to the individual scratchpads, resulted in a power consumption of 150mW, and with the output stationary method, the dynamic power consumption was around 117mW. However, in the RS dataflow method with an efficient filter reuse strategy, the dynamic power requirement was 45mW. Hence, RS data flow uses only 30% power of the weight stationary and 38% power of the output stationary dataflow. Here, one row of filter weights and one row Ifmaps generates one row of PSums, and each row pair stays stationary in the individual PE. For the continuous iteration of each layer, one row of filter weights is kept static in the PE, and the Ifmaps are streamed in from the scratchpad to reduce the bandwidth access. During each layer iteration, the scratchpad size mainly depends on the filter size of the layer and not on the Ifmaps size or the PSums size. For example, in AlexNet, for CONV1 layer, the scratchpad size is 11, for CONV2, it is 5, and for CONV3-CONV5, it is 3. The same row of filter weights is shared across all the PEs horizontally in a two-dimensional convolution using the filter reuse strategy. The PSums from all the PEs are finally accumulated together. Reusing the filter weights avoids unnecessary data movement and helps improve the processor's energy efficiency. The filter reuse technique has higher benefits, especially in FC operations, as the order of weights is substantial for computation.

The computation unit consists of the PE elements arranged in a systolic array [126], where the Ifmaps are accessed from the top, and weight data are accessed horizontally. With 1024 elements, a 1-D convolution is aggregated to perform a 2-D convolution. Each PE consists of an N-bits approximate multiplier and an adder unit arranged in a pipelined manner. Each Ifmaps is accumulated and multiplied with the weight data once. Rather than performing

the MAC with every new Ifmaps, we term this Accumulate-Multiply (ACM) technique. This technique reduces the dynamic power consumption by around 30%

The two leading detectors in the architecture are mainly used to locate the most significant '1' in each of the respective N-bits operands. The most significant '1' location selects the following N-2 consecutive bits on the desired accuracy. For example, if the input to the leading one detector (LOD) is 16'b0010_1111_1010_1111, then the output of the circuit will be 16'b0010_0000_0000_0000. The output of LOD is priority encoded to generate the log(n) bits. The two multiplexers are responsible for selecting N-2 bits from each operand. Starting from the most significant '1', the barrel shifter shifts the multiplication output to the right to generate the 2N bit results. Finally, the 2N bits are reduced to N-bits by categorically truncating based on the fixed-point q-format. The accumulated data will be multiplied with weight data to generate PSums results. After the first activation is performed in CONV1 layer, most of the PSums results will be sparse. To avoid unnecessary computation of sparse results, a zero-gating logic is provided at the multiplier input to detect if the N-2 bits of the IF are zero. If the results are sparse, the multiplication operation is avoided to reduce the dynamic power consumption. The zero-gating logic and the approximate multiplier save up to 25% of dynamic power consumption.

4.2.4 Other System Modules

The accumulator array adds the PSums from the PE array with the bias data to generate the final convolution and FC results. The same non-linear activation function ReLU ($f(x) = max(0, x)$) is used for the sake of uniformity in the inference of different DNN models. The pooling module is used to obtain the maximum of the matrix. Truncator block truncates the pooling data for CONV1, CONV2, CONV5 and ReLU data for CONV3, CONV4 layers of the AlexNet. Finally, the truncated results are stored in the PSums memory to compute the next layer. During the next cycle of operation, PSums results from the previous layer are calculated with the subsequent layer weights of the same bit-width pattern as that of the truncated output of the previous layer.

4.3 Evaluation

4.3.1 Comparison Metric

To observe the influence of bit width on the overall response of the network, we evaluated the network output after each layer. The output of a convolution layer is a 3D tensor. Each 2D slice (matrix) of this 3D tensor corresponds to the output of a particular kernel for that layer. The output 2D matrix is compared with the reference output of that kernel. The TensorFlow implementation of the AlexNet[1] is used to obtain the reference output. For each matrix, root means square error (RMSE) is computed according to Eq. 4.1.

$$ConvE_k^l = \sqrt{\left(\frac{\sum_{r=1}^{n_r}\sum_{c=1}^{n_c}\left(F_l(r,c,k) - R_l(r,c,k)\right)^2}{n_r \times n_c}\right)} \tag{4.1}$$

[1] https://www.cs.toronto.edu/~guerzhoy/tf_alexnet/

$ConvE_k^l$ and FcE^l are calculated $\forall\ k \in \{1, \ldots, n_k\}$, where F_l, R_l denote the 3D tensor of FPGA output and reference output, respectively for any particular layer l and (r, c, k), are row, column, and channels respectively. For example, in AlexNet, the output of the first convolutional layer ($l = 1$) is of size $55 \times 55 \times 96$ ($n_r \times n_c \times n_k$). Comparison performed on this layer would result in 96 element array E^1. Unlike convolutional layers, the output of a FC layer is the 1-D array for which RMSE is calculated according to the Eq. 4.2. For the last FC layer (FC8), the accuracy is shown in the form of top-1 and top-5 accuracy.

$$FcE^l = \sqrt{\left(\frac{\sum_{k=1}^{n_k} \left(F_l(k) - R_l(k)\right)^2}{n_k}\right)} \tag{4.2}$$

Let us consider a simple example in the first convolution layer of the AlexNet, where the 3D tensor output from the FPGA after the first inference is 220, and the reference output is 210. The n_r is 55, the n_c is 55, the final RMS error $convE_k^l$ is 0.181 as shown in Eq. 4.3. The FPGA outputs and the reference output are obtained after computing all the rows, columns, and channels of the first layer of the AlexNet. Similarly, the RMS error for the FC layer is also computed.

$$convE_k^l = \sqrt{\frac{(220 - 210)^2}{55 \times 55}} \tag{4.3}$$

4.3.2 Methodology

The proposed architecture was implemented using System Verilog, and behavioral simulations were performed using a commercial Verilog simulator. For each layer of the network, a text file of the inference output was generated, which in turn was compared with the reference output of the trained ideal DNN to determine the RMS error according to Eq. 4.1 or 4.2. The synthesis, implementation of the architecture, and calculation for the power consumption, timing, and hardware resource usage were performed using Xilinx Vivado. Ten different random images from the web were used to evaluate the architecture.

The architecture was synthesized using the Synopsys Design Compiler (DC) for the ASIC implementation using the GF 22nm FDSOI CMOS technology. The design was placed and routed using Synopsys IC Compiler2 (ICC2). After the post-layout extraction, the timing and power sign-off were performed using Prime Time.

4.4 Experimental Results

One main objective of this work is to propose an advanced truncation strategy, i.e. hardware optimization approaches to perform a multi-bit operation on the inferences of DNN models for this accelerator. In the brief overview of this section, we will report the resource utilization and power consumption of each module of the DDR3 and MIG-7 memory controller. Secondly, the different fixed bit widths of various DNN models are compared based on the RMS error, resource utilization, and power consumption. Then, a truncation of the DNN models is presented. Different sets of multi-bit analysis experiments were performed based on the truncation outcome. Finally, we conclude with the best truncation combination with the lowest

Table 4.2: Resource utilization and power consumption of DDR3-MIG7 memory controller on the Virtex Ultrascale FPGA [31] ©2021 IEEE.

Modules	BR	LUT	FF	DSP	P (W)
MIG-7	0	12902	11056	0	1.691
Proc Sys Rst	0	18	38	0	0.001
Micoblaze Debugger	0	91	110	0	0.001
Microblaze	19	2559	2062	5	0.086
AXI_Int_Connect_1	0	397	348	0	0.002
Local Mem. Hierarchy	2	12	14	0	0.001
AXI_Int_Connect_2	0	115	113	0	0.001
AXI_BRAM_Ctrl	0	269	243	0	0.002
AXI_timer	0	293	240	0	0.001
AXI_int_cntrl	0	59	62	0	0.001
AXI_uartlite	0	100	109	0	0.001
Block Mem. Gen	2	10	10	0	0.003
Total	**23**	**16825**	**14405**	**5**	**1.791**

RMS error and higher accuracy suitable for the multi-bit accelerator and present its results for different DNN models.

4.4.1 Resource Utilization and Power Consumption of DDR3-MIG7 Memory Controller

The resource utilization and power consumption of the DDR3-MIG7 memory controller are shown in Table 4.2. The memory interface generator consumes the highest resources and power, accounting for 77%. From Table 4.2, it is evident that our architecture consumes less than 1% of comprehensive resources and a meager amount of power on the FPGA due to the efficient implementation of the communication interface protocol between the on-chip and off-chip memories.

4.4.2 Fixed Bit Architecture

This sub-section presents the results for different bit widths starting from 32 bits to 8 bits. Each bit was inference for a hundred different images, and an average of their resource utilization and power consumption is shown in Table 4.4–4.7 for AlexNet, MobileNet, SqueezeNet, and EfficientNet respectively. As expected, 32-bits have the highest resource utilization, and the 8-bits have the least. From Table 4.4, we can also observe that the resource utilization of FC layers is more than the preceding CONV layers due to its enormous weight data and the PSums results. From each of the tables, it can be observed that the power consumption is relatively lower when compared to our previous work [30]. PE is now performed using the ACM technique instead of the MAC method previously used. A MAC technique for a 1024 PE array consumes around 18W. However, an ACM technique and approximate multiplication consume around 5.8W. Hence, there is only 30% of the previous power requirement due to

Table 4.3: Mean error for different bit width for all layers of AlexNet [31] ©2021 IEEE.

Layers	32-Bits	28-Bits	24-Bits	16-Bits	8-Bits
Conv1	0.02	0.05	0.73	69.92	97.46
Conv2	0.04	0.18	5.80	125.64	134.83
Conv3	0.06	0.25	9.96	119.10	118.78
Conv4	0.05	0.20	7.18	73.85	73.76
Conv5	0.04	0.14	4.74	45.53	45.60
FC6	0.04	0.14	2.35	21.68	21.76
FC7	0.06	0.09	0.77	6.24	7.16
FC8	0.09	0.22	1.42	17.99	18.05

the ACM technique. In Table 4.4, the resource utilization and the power consumption for the FPGA are from the largest layers for each DNN model.

For the 32-bit analysis in the CONV1 layer of AlexNet, the total number of DSPs is 2714, and the corresponding peak performance throughput is 0.18GHz $\times$ 2714 $\times$ 2 = 977.04 GOPS. The value “2" in the performance throughput signifies multiplication and addition operations. Similarly, in the FC6 layer, the total DSPs were 2266, and the peak performance was 815.76 GOPS. With the decrease in bit width, the peak performance also reduces in all the layers of the different DNN models, as shown in Fig. 4.6. For the fixed bit architecture, the mean RMS error (according to Eq. 4.1 and Eq. 4.2) for different layers of the AlexNet is shown in Table 4.3. When the bit width is reduced, the RMS error increases and goes beyond 100% if there is a vast difference between the inference and the reference results.

The difference is also due to scaling from the floating-point during the training phase to the fixed-point during the inference phase and truncation in the integer and fractional parts.

In Table 4.4-4.7, the power consumption and resource utilization will not scale proportionately as they depend on multiple factors like the layers under the inference, the truncation performed on the particular layer, the sparsity from the Ofmaps of each layer and the static power consumption of the device under inference. The number of channels and the filter size will decide the power consumption. With smaller filter sizes and shorter strides, more MAC operations are required to complete the evaluation, increasing power consumption.

From the above experiments, with the decrease in the bit-width, the RMS error does not increase significantly due to the pertinent truncation process, as the bits are not truncated randomly but in the proper scheme as explained in the next subsection 4.4.3. So, if we truncate the previous layer's output before feeding it into the next layer, we could achieve improved performance with an acceptable reduction in RMS error. The truncation can be performed either in the integer or fractional part. In the following subsection 4.4.3, we explore different bit arrangements and their response to overall performance and power consumption.

4.4.3 Truncation Loss

It can be truncated before feeding the previous layer's output to the next layer's input. Truncation is usually performed on the LSB of the fractional part, and MSB is avoided as the RMS error is relatively high. In the integer part, both MSB and LSB are sequentially

Table 4.4: Resource utilization (BR: BRAM, FF, LUT, DSP) in % and power (P) in W of fixed bit for AlexNet, measured on Virtex Ultrascale FPGA[30] [31] ©2021 IEEE.

Layers	32-Bits - 1_15_16					28-Bits-1_13_14					24-Bits-1_11_12					16-Bits-1_7_8					8-Bits-1_3_4				
	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P
Conv1	78	62	76	94	11.4	74	59	73	92	11.1	67	56	68	89	10.5	59	51	63	83	9.1	54	46	60	80	9.9
Conv2	33	26	28	78	11.1	32	24	28	74	10.9	35	24	27	72	10.3	30	22	24	69	8.7	27	22	24	65	7.2
Conv3	27	19	21	73	11.1	27	19	20	70	10.8	28	19	20	68	9.9	22	18	18	64	8.2	20	18	18	61	6.9
Conv4	28	19	21	71	11.3	28	19	20	66	11.1	27	19	20	64	9.5	22	18	18	61	7.7	19	18	18	59	6.5
Conv5	26	19	21	69	11.0	24	19	20	65	10.8	27	19	20	62	9.8	21	18	17	59	7.8	18	18	17	56	7.0
FC6	28	41	43	79	12.8	28	28	40	78	12.5	29	28	40	76	12.0	23	24	36	74	9.4	21	23	34	71	7.9
FC7	21	37	36	75	12.1	20	33	36	72	12.0	22	31	34	70	11.4	16	26	33	66	8.7	13	19	28	63	6.9
FC8	18	33	34	73	11.5	17	32	33	68	10.9	17	29	31	63	10.1	14	23	28	62	7.4	12	19	21	52	3.9

Table 4.5: Resource utilization (BR: BRAM, FF, LUT, DSP) in % and power (P) in W of fixed bit for MobileNet, measured on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	32-Bits - 1_15_16					28-Bits-1_13_14					24-Bits-1_11_12					16-Bits-1_7_8					8-Bits-1_3_4				
	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P
Conv1	76	66	80	88	12.1	71	61	75	83	11.6	69	57	73	81	11.2	64	54	68	77	7.9	60	51	65	73	7.6
Conv2	86	86	85	90	11.8	81	84	82	88	11.5	85	81	80	87	11.0	73	78	76	83	7.6	68	74	73	87	7.4
Conv3	84	87	87	92	11.8	80	82	84	91	11.3	83	78	82	92	10.6	71	74	77	89	7.0	66	69	71	84	6.5
Conv4	82	91	84	94	12.0	79	84	81	92	11.7	78	78	78	90	10.2	70	71	72	86	6.7	62	66	68	81	6.1
Conv5	80	86	95	93	11.7	78	82	92	88	11.4	73	77	88	86	10.5	68	72	83	82	6.8	59	66	80	76	5.9
Conv6	83	92	93	96	13.6	80	84	84	93	13.1	76	78	82	89	12.8	71	74	76	86	9.1	63	72	71	81	5.2
Conv7	79	84	88	94	12.8	81	81	81	93	12.6	72	77	78	87	12.2	70	72	74	82	8.9	62	67	67	78	4.4
Conv8	78	82	85	97	12.2	79	78	76	95	11.5	70	72	75	90	10.8	68	66	72	79	7.4	60	61	65	73	4.0
Conv9	77	81	85	98	12.1	76	78	75	95	11.7	68	73	74	88	10.9	63	67	68	83	7.6	57	61	64	80	4.3
Conv10	77	79	84	97	12.2	74	79	75	92	11.9	67	72	76	91	11.8	61	68	72	84	8.2	59	63	67	76	5.2
Conv11	76	78	83	96	12.1	73	76	73	89	11.9	66	72	70	87	11.6	60	63	65	81	8.1	57	58	63	71	5.0
Conv12	72	69	67	77	12.3	69	66	64	74	12.1	55	59	62	68	11.8	58	56	56	63	8.4	54	53	54	61	5.4
Conv13	71	67	64	72	12.1	66	62	62	66	11.7	51	58	57	63	11.3	54	54	52	59	7.8	51	49	47	54	5.0
Conv14	64	66	57	68	11.5	61	61	55	63	11.1	46	56	51	59	10.7	52	52	46	54	7.4	48	47	42	50	4.7
FC15	58	61	51	62	11.3	54	56	47	59	10.9	50	53	43	56	10.5	48	43	38	49	7.0	42	41	36	47	4.1

Table 4.6: Resource utilization (BR: BRAM, FF, LUT, DSP) in % and power (P) in W of fixed bit for SqueezeNet, measured on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	32-Bits - 1_15_16					28-Bits-1_13_14					24-Bits-1_11_12					16-Bits-1_7_8					8-Bits-1_3_4				
	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P
Conv1	76	66	80	88	12.06	71	61	75	83	11.56	69	57	73	81	10.45	64	54	68	77	8.30	60	51	65	73	6.87
Fire 2	56	61	73	83	8.83	53	57	71	80	8.26	51	52	67	76	7.81	47	47	65	73	6.54	43	44	62	70	5.69
Fire 3	56	61	73	83	8.83	53	57	71	80	8.26	51	52	67	76	7.81	47	47	65	73	6.54	43	44	62	70	5.69
Fire 4	69	68	78	86	8.98	66	66	77	83	8.38	61	63	74	81	7.90	58	59	72	78	6.60	56	56	70	74	5.73
Fire 5	51	63	75	83	8.76	48	59	71	80	8.21	46	56	66	77	7.77	42	53	63	73	6.51	39	51	58	70	5.67
Fire 6	53	67	80	86	8.95	51	62	75	83	8.36	48	58	71	80	7.89	46	54	50	77	6.59	43	49	46	73	5.72
Fire 7	53	67	80	86	8.95	51	62	75	83	8.36	48	58	71	80	7.89	46	54	50	77	6.59	43	49	46	73	5.72
Fire 8	59	72	85	88	8.98	57	68	81	87	8.38	53	66	77	83	7.90	51	63	75	80	6.66	47	63	54	76	5.77
Fire 9	50	61	73	81	8.57	47	58	70	78	8.06	43	56	66	74	7.65	40	52	63	71	6.43	37	48	58	67	5.62
Conv10	52	67	77	87	8.66	48	62	74	83	8.13	46	58	71	80	7.70	43	59	67	77	6.46	40	57	64	73	5.64

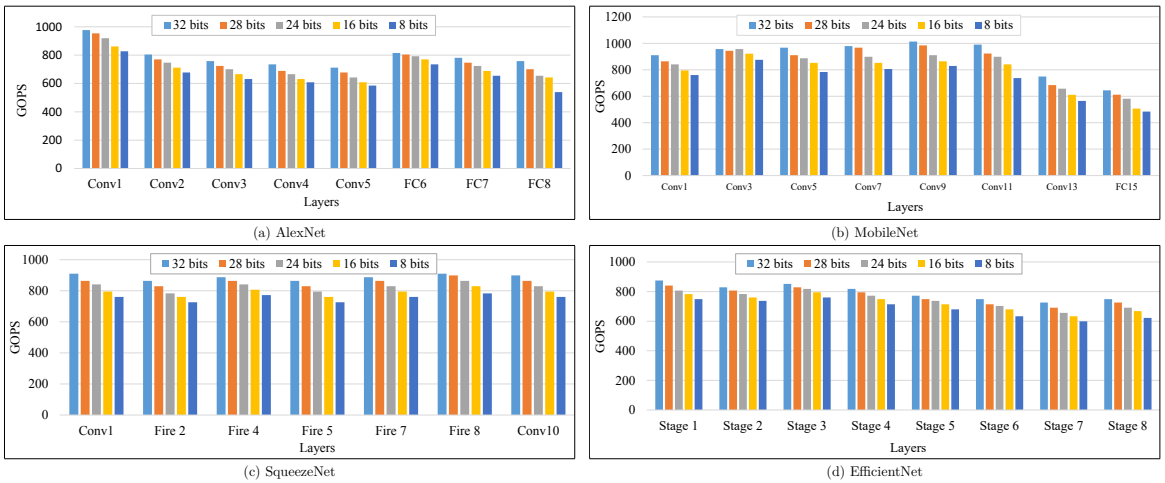
Table 4.7: Resource utilization (BR: BRAM, FF, LUT, DSP) in % and power (P) in W of fixed bit for EfficientNet, measured on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	32-Bits - 1_15_16					28-Bits-1_13_14					24-Bits-1_11_12					16-Bits-1_7_8					8-Bits-1_3_4				
	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P	BR	FF	LUT	DSP	P
Stage 1	83	72	85	84	13.1	80	68	82	81	11.7	78	66	78	78	10.6	74	63	76	76	8.4	73	61	73	72	6.9
Stage 2	78	68	73	80	12.8	76	66	71	78	11.4	73	63	67	76	10.3	71	61	66	73	8.2	69	58	63	71	6.8
Stage 3	80	71	76	82	12.9	78	68	74	80	11.5	77	66	71	79	10.4	72	64	68	77	8.3	70	59	66	73	6.9
Stage 4	78	67	74	79	12.8	74	64	70	77	11.4	72	62	66	74	10.3	70	58	63	72	8.2	67	54	60	69	6.8
Stage 5	73	62	71	74	12.0	71	59	68	72	10.8	69	57	63	71	9.8	64	54	60	69	7.9	61	51	57	66	6.6
Stage 6	70	58	66	72	11.7	68	56	65	69	10.6	66	51	61	68	9.7	62	49	57	66	7.8	59	46	53	61	6.5
Stage 7	68	54	62	70	11.6	66	52	60	67	10.5	62	48	58	63	9.6	59	46	54	61	7.7	54	42	51	58	6.5
Stage 8	71	58	66	72	12.1	70	56	64	70	10.9	66	53	61	67	9.9	62	48	58	64	7.9	58	44	54	60	6.6

Table 4.8: Truncation results for the different bit combinations of integer and fractional values for different conv layers of AlexNet[30] [31] ©2021 IEEE.

Conv1-Conv2					Conv2-Conv3					Conv3-Conv4					Conv4-Conv5				
Input Comb	Truncation			RMS Error	Input Comb	Truncation			RMS Error	Input Comb	Truncation			RMS Error	Input Comb	Truncation			RMS Error
	MSB Int	LSB Int	LSB Frac			MSB Int	LSB Int	LSB Frac			MSB Int	LSB Int	LSB Frac			MSB Int	LSB Int	LSB Frac	
15_16	**0**	**0**	**0**	**0.018**	15_16	**0**	**0**	**0**	**0.034**	15_16	**0**	**0**	**0**	**0.058**	15_16	**0**	**0**	**0**	**0.038**
	2	**0**	**2**	**0.06**		**2**	**0**	**2**	**0.051**	12_11	**0**	**0**	**0**	**0.058**	13_14	**2**	**0**	**2**	**0.218**
	0	2	2	142.25		4	0	0	6.742	13_14	**0**	**0**	**0**	**0.06**		1	0	3	0.214
	1	0	3	0.057		0	0	4	0.06		2	0	2	32.245		0	0	4	0.214
	0	1	3	93.48		0	0	8	0.061		0	2	2	118.145		**1**	**0**	**7**	**1.645**
	4	0	0	6.56		**3**	**0**	**5**	**0.054**		1	0	3	3.897	12_11	2	0	6	3.587
	0	4	0	177.6		2	0	6	0.06	12_11	0	1	3	78.542		3	0	5	27.142
	0	0	4	0.05		**1**	**0**	**3**	**0.176**		4	0	0	117.654		**1**	**0**	**3**	**18.546**
	3	0	1	0.05	13_14	0	0	4	0.176		0	4	0	146.82	12_7	0	0	4	18.546
	0	3	1	165.23		2	0	2	20.146		**0**	**0**	**4**	**1.896**		2	0	2	19.412

truncated. Removing MSB from the fractional part would significantly affect the fractional value, which may be undesirable in some cases. Therefore, a majority of the calculations are



(a) AlexNet (b) MobileNet (c) SqueezeNet (d) EfficientNet

Figure 4.6: Peak performance of different layers for fixed bit combinations for different networks [31] ©2021 IEEE.

Table 4.9: Truncation results for the different bit combinations of integer and fractional values for CONV5 and FC6 layers of AlexNet [30] [31] ©2021 IEEE.

Conv5-FC6				
Input Comb	Truncation			RMS Error
	MSB Int	LSB Int	LSB Dec	
15_16	**0**	**0**	**0**	**0.029**
11_12	**0**	**0**	**0**	**0.214**
11_4	**0**	**0**	**0**	**1.645**
	2	0	2	76.412
	0	2	2	65.982
	1	0	3	76.412
	0	1	3	65.874
	4	0	0	67.142
	0	**4**	**0**	**67.142**
	0	0	4	67.965

Table 4.10: Truncation results for the different bit combinations of integer and fractional values for FC layers of AlexNet [30] [31] ©2021 IEEE.

FC6-FC7					FC7-FC8				
Input Comb	Truncation			RMS Error	Input Comb	Truncation			RMS Error
	MSB Int	LSB Int	LSB Dec			MSB Int	LSB Int	LSB Dec	
15_16	**0**	**0**	**0**	**0.04**	15_16	**0**	**0**	**0**	**0.03**
11_4	**0**	**0**	**0**	**1.68**	11_4	**0**	**0**	**0**	**10.45**
11_12	**2**	**0**	**2**	**0.645**	9_10	**2**	**0**	**2**	**0.684**
	1	0	3	0.65		0	0	4	0.678
	0	0	4	0.65		1	0	3	0.678
	3	0	1	3.145		0	1	3	5.46
7_4	**0**	**2**	**2**	**33.412**		3	0	1	3.266
	1	0	3	33.412		0	3	1	9.191
	0	1	3	33.412		0	2	2	7.944
	4	0	0	34.956	3_4	**4**	**0**	**0**	**34.965**

performed in the fractional part. Based on the above hypothesis, we performed truncation between consecutive pairs of layers, as shown in Table 4.8. *"Input comb"* denotes the bit

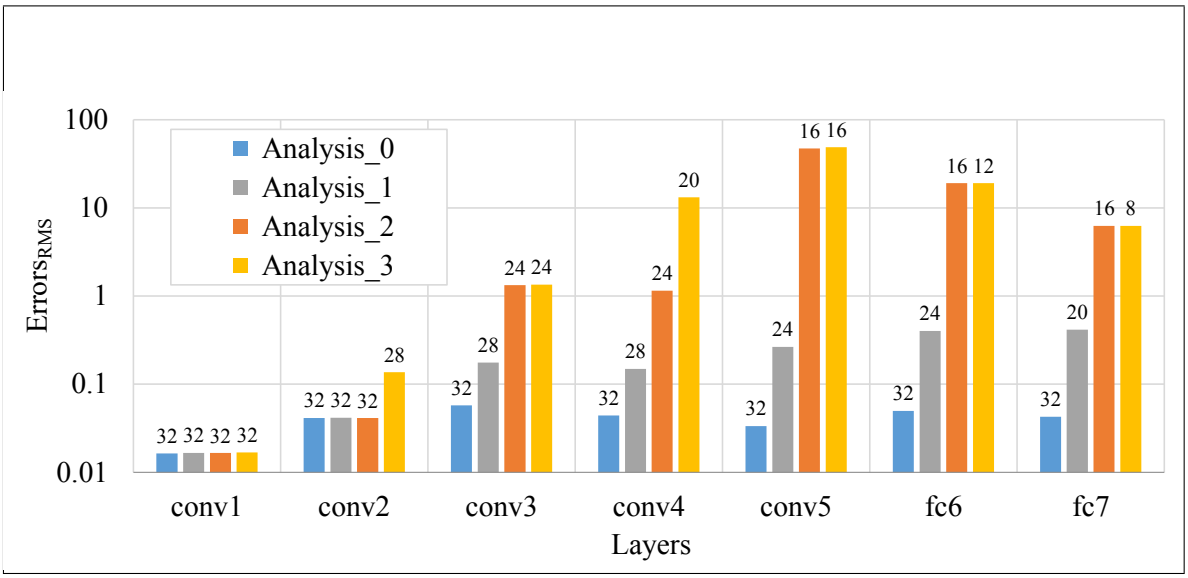


Figure 4.7: RMS error for different layers of AlexNet for various bit combinations. The bit width of layers is shown as data labels [30] [31] ©2021 IEEE.

Table 4.11: RMS error results for all the layers of MobileNet based on analysis 1 [31] ©2021 IEEE.

Layers	Input Comb	Truncation			RMS Error
		MSB Int	LSB Int	LSB Frac	
Conv1	15_16	0	0	0	0.025
Conv2		0	0	0	0.028
Conv3		0	0	0	0.031
Conv4		2	0	2	0.272
Conv5	13_14	0	0	0	0.276
Conv6		0	0	0	0.286
Conv7		2	0	2	0.56
Conv8	11_12	0	0	0	0.568
Conv9		0	0	0	0.587
Conv10		0	0	0	0.588
Conv11		2	0	2	0.972
Conv12	9_10	0	0	0	0.986
Conv13		0	0	0	0.986
Conv14		0	0	0	0.99

arrangement of the output of the previous layer. For example, in the first pair: CONV1 - CONV2, the output of CONV1 is 32 bits with a bit arrangement of 1_15_16 where 1, 15, and 16 represent the sign, integer, and fractional bit width, respectively. For simplicity, the sign part, which is fixed, therefore, is not shown. The truncation is performed in MSB/LSB of an integer and LSB of a fractional part. Each pair explores ten truncations, and the corresponding RMS error is computed. In the first pair CONV1 - CONV2 of Table 4.8, the input combinations were 32 32-bits, and we truncated them into 28 bits before performing the CONV2 operations. The MSB int indicates the truncation of the bits from the MSB on the integer part, LSB int indicates the truncation of the bits from the LSB on the integer part, and LSB dec indicates the truncation of the bits from the LSB on the decimal part.

Table 4.12: RMS error results for all the layers of EfficientNet based on analysis 1 [31] ©2021 IEEE.

Layers	Input Comb	Truncation			RMS Error
		MSB Int	LSB Int	LSB Frac	
Stage 1	15_16	0	0	0	0.046
Stage 2		2	0	2	0.149
Stage 3	13_14	0	0	0	0.156
Stage 4		2	0	2	0.305
Stage 5	11_12	0	0	0	0.315
Stage 6		2	0	2	0.45
Stage 7	9_10	0	0	0	0.488
Stage 8		0	0	0	0.489

Table 4.13: RMS error results for all the layers of SqueezeNet based on analysis 1 [31] ©2021 IEEE.

Layers	Input Comb	Truncation			RMS Error
		MSB Int	LSB Int	LSB Frac	
Conv1	15_16	0	0	0	0.035
Fire 2		0	0	0	0.038
Fire 3		2	0	2	0.265
Fire 4	13_14	0	0	0	0.279
Fire 5		0	0	0	0.279
Fire 6		2	0	2	0.671
Fire 7	11_12	0	0	0	0.687
Fire 8		2	0	2	0.887
Fire 9	9_10	0	0	0	0.896
Conv10		0	0	0	0.896

Here, we have not included the truncation of the bits from the MSB on the decimal part, as it is not feasible and leads to higher RMS errors. For example: From Table 4.8 the second combination of the first pair, we have truncated 2 bits from the MSB of the integer part, 0 bits from the LSB of the integer part, and 2 bits from the LSB of the decimal part. So, the output combination of bits will be 1_13_14, which is 28 bits, and its RMS error is according to Eq. 4.1. The bolded values indicate the selected truncation combination of a pair, which also serves as the input combination for the next pair. In Table 4.9, we present the truncation results of the last convolution layer CONV5 and the first FC layer FC6. Table 4.10 shows the results of FC6-FC7 and FC7-FC8 pairs, where the RMS error is calculated according to Eq. 4.2. These truncation results with its MSB/LSB combination and errors were consistent when explored on the other DNN models.

4.4.4 Multi-Bit Architecture

From the performance of the truncation, we were motivated to explore the reduction in bits systematically. From the pairs highlighted in Table 4.8, 4.9, 4.10, we created four sets of analyses which were explicitly evaluated. Table 4.14, Table 4.15, Table 4.16 and Table 4.17

Table 4.14: Resource utilization in % and power (P) in W for different bit combinations for AlexNet on Virtex Ultrascale FPGA [30] [31] ©2021 IEEE.

Layers	Analysis 0						Analysis 1						Analysis 2						Analysis 3					
	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P
Conv1	15_16	78	62	76	94	11.38	15_16	78	62	76	94	11.38	15_16	78	62	76	94	11.38	15_16	78	62	76	94	11.38
Conv2		33	26	28	78	11.11		33	26	28	78	11.11		33	26	28	78	11.11	13_14	32	24	28	74	10.95
Conv3		27	19	21	73	11.13	13_14	27	19	20	70	10.75	12_11	28	19	20	68	9.94	12_11	28	19	20	68	9.94
Conv4		28	19	21	71	11.29		28	19	20	66	11.11		27	19	20	64	9.54	12_7	25	18	19	62	8.76
Conv5		26	19	21	69	11.02	11_12	27	19	20	62	9.83		21	18	17	59	7.82	11_4	21	18	17	59	7.82
FC6		28	41	43	79	12.82		29	28	40	76	12.01	11_4	23	24	36	74	9.36	7_4	17	14	14	54	6.98
FC7		21	37	36	75	12.10	9_10	15	26	28	59	9.97		16	26	33	66	8.69	3_4	13	19	28	63	6.88

Table 4.15: Resource utilization in % and power (P) in W for different bit combinations for MobileNet on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	Analysis_0						Analysis_1						Analysis_2						Analysis_3					
	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P
Conv1	15_16	76	66	80	88	12.06	15_16	76	66	80	88	12.06	15_16	76	66	80	88	12.06	15_16	76	66	80	88	12.06
Conv2		86	86	85	90	11.78		86	86	85	90	11.78		86	86	85	90	11.78		86	86	85	90	11.78
Conv3		84	87	87	92	11.80		84	87	87	92	11.80		84	87	87	92	11.80	13_14	80	82	84	91	11.29
Conv4		82	91	84	94	11.97		82	91	84	94	11.97		82	91	84	94	11.97		79	84	81	92	11.66
Conv5		80	86	95	93	11.68		78	82	92	88	11.37		80	86	95	93	11.68	12_11	73	77	88	86	10.48
Conv6		83	92	93	96	13.59	13_14	80	84	84	93	13.13		76	78	82	89	12.81		76	78	82	89	12.81
Conv7		79	84	88	94	12.83		81	81	81	93	12.59		72	77	78	87	12.17	12_7	70	74	76	84	11.70
Conv8		78	82	85	97	12.21		70	72	75	90	10.79	12_11	70	72	75	90	10.79		67	69	72	88	10.50
Conv9		77	81	85	98	12.13	11_12	68	73	74	88	10.90		68	73	74	88	10.90	11_4	63	67	68	83	7.60
Conv10		77	79	84	97	12.18		67	72	76	91	11.79		67	72	76	91	11.79		61	68	72	84	8.16
Conv11		76	78	83	96	12.10		66	72	70	87	11.65		60	63	65	81	8.05	7_4	58	62	63	78	7.70
Conv12		72	69	67	77	12.25	9_10	55	59	62	68	9.65	11_4	58	56	56	63	8.44		55	55	53	61	7.90
Conv13		71	67	64	72	12.12		48	57	61	61	9.24		54	54	52	59	7.84	3_4	51	49	47	54	4.98
Conv14		64	66	57	68	11.47		44	54	57	58	8.71		52	52	46	54	7.40		48	47	42	50	4.68

Table 4.16: Resource utilization in % and power (P) in W for different bit combinations for SqueezeNet on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	Analysis 0						Analysis 1						Analysis 2						Analysis 3					
	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P
Conv1	15_16	76	66	80	88	12.1	15_16	76	66	80	88	12.1	15_16	76	66	80	88	12.1	15_16	76	66	80	88	12.1
Fire 2		56	61	73	83	8.8		56	61	73	83	8.8		56	61	73	83	8.8		56	61	73	83	8.8
Fire 3		56	61	73	83	8.8		56	61	73	83	8.8		56	61	73	83	8.8	13_14	53	57	71	80	8.3
Fire 4		69	68	78	86	9.0		66	66	77	83	8.4		69	68	78	86	9.0		66	66	77	83	8.4
Fire 5		51	63	75	83	8.8	13_14	48	59	71	80	8.2		46	56	66	77	7.8	12_11	46	56	66	77	7.8
Fire 6		53	67	80	86	9.0		51	62	75	83	8.4	12_11	48	58	71	80	7.9		48	58	71	80	7.9
Fire 7		53	67	80	86	9.0	11_12	48	58	71	80	7.9		48	58	71	80	7.9	12_7	47	56	68	75	6.9
Fire 8		59	72	85	88	8.98		53	66	77	83	7.9		51	63	75	80	6.7		49	62	73	76	7.1
Fire 9		50	61	73	81	8.57	9_10	40	53	62	72	7.43	11_4	40	52	63	71	6.4	11_4	40	52	63	71	6.4
Conv10		52	67	77	87	8.66		44	54	67	77	7.58		43	59	67	77	6.5		43	59	67	77	6.5

Table 4.17: Resource utilization in % and power (P) in W for different bit combinations for EfficientNet on Virtex Ultrascale FPGA [31] ©2021 IEEE.

Layers	Analysis 0						Analysis 1						Analysis 2						Analysis 3					
	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P	Bits	BR	FF	LUT	DSP	P
Stage 1	15_16	83	72	85	84	13.1	15_16	83	72	85	84	13.1	15_16	83	72	85	84	13.1	15_16	83	72	85	84	13.1
Stage 2		78	68	73	80	12.8		78	68	73	80	12.8		78	68	73	80	12.8		78	68	73	80	12.8
Stage 3		80	71	76	82	12.9	13_14	78	68	74	80	11.5		80	71	76	82	12.9	13_14	78	68	74	80	11.5
Stage 4		78	67	74	79	12.8		74	64	70	77	11.4		72	62	66	74	10.3		74	64	70	77	11.4
Stage 5		73	62	71	74	12.0	11_12	69	57	63	71	9.8	12_11	69	57	63	71	9.8	12_11	69	57	63	71	9.8
Stage 6		70	58	66	72	11.7		66	51	61	68	9.7		66	51	61	68	9.7		66	51	61	68	9.7
Stage 7		68	54	62	70	11.6	9_10	62	48	58	63	9.6	11_4	59	46	54	61	7.7	12_7	62	49	76	65	7.9
Stage 8		71	58	66	72	12.1		65	51	60	67	9.9		62	48	58	64	7.9	11_4	62	48	54	64	7.9

lists all the four analyses with the bit combinations, followed by resource utilization and power consumption for all the four DNN models. For comparison, we created a reference set with 32 bits called Analysis_0. Peak performance for different analyses for different DNN models is shown in Fig. 4.8.

The corresponding RMS error for these analyses was also computed and is shown in Fig. 4.7 for AlexNet. For RMS errors, a logarithmic scale was used to visualize significant variations. Analysis_1 had a better trade-off between accuracy, power, and resource utilization on the Virtex Ultrascale FPGA among all the different bit combinations. So, based on the Analysis_1, we have reported the RMS error results for MobileNet, EfficientNet, and SqueezeNet in Table 4.11, Table 4.12 and Table 4.13 respectively.

The last layer of each DNN model is the classification layer on which the softmax function is applied. Extending Analysis_1, we further tried to reduce the bit width of the last layer

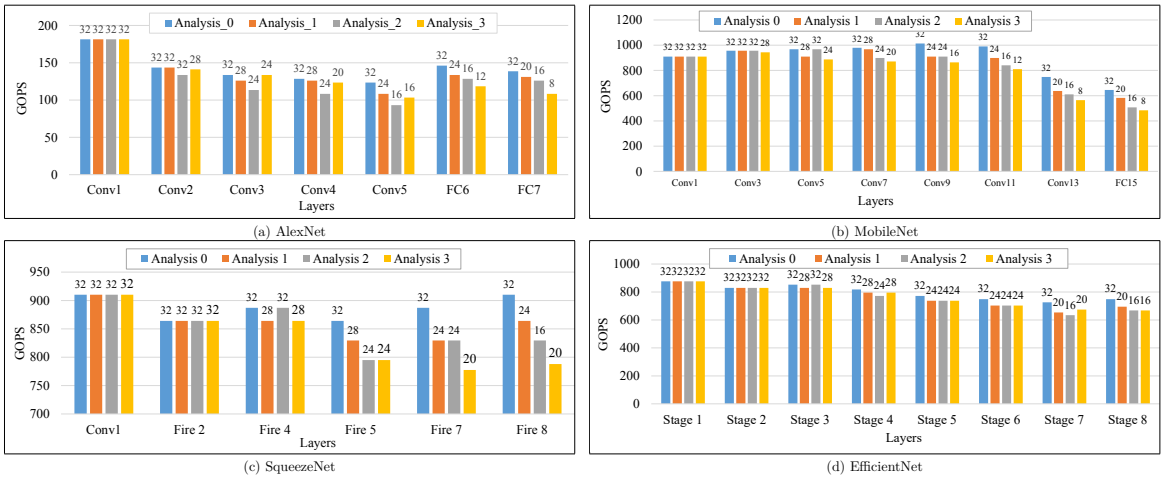


Figure 4.8: Peak performance of different layers for multi-bit combinations for different networks [31] ©2021 IEEE.

Table 4.18: Resource utilization in %, power measured (P) in W and peak performance (PP) in GOPS for last layers measured on Virtex Ultrascale FPGA [30] [31] ©2021 IEEE.

	AlexNet							MobileNet						SqueezeNet						EfficientNet					
Bit Width	Bits	BR	FF	LUT	DSP	P	PP	BR	FF	LUT	DSP	P	PP	BR	FF	LUT	DSP	P	PP	BR	FF	LUT	DSP	P	PP
20	9_10	19	23	31	66	7.37	167.44	50	53	43	56	10.5	141.12	45	62	73	79	7.1	199.08	65	51	60	67	9.92	168.84
19	9_9	18	23	31	65	7.12	164.64	49	50	42	54	10.26	136.08	45	61	72	78	6.9	196.56	64	52	59	65	9.4	163.8
18	8_9	18	22	30	64	6.9	161.84	49	47	42	53	9.8	133.56	44	61	70	78	6.78	196.56	64	50	59	65	8.8	163.8
17	8_8	17	22	30	64	6.79	161.84	48	45	40	50	9.27	126	44	60	68	77	6.65	194.04	63	49	58	64	8.34	161.28
16	7_8	17	22	28	63	6.17	159.04	48	44	38	49	8.5	123.2	43	59	67	77	6.5	193.2	62	48	58	64	7.9	162.4
15	7_7	15	22	27	62	5.95	156.24	47	43	38	48	7.6	120.96	42	58	66	76	6.3	191.52	62	48	58	63	7.6	158.76
14	7_6	15	21	27	60	5.63	150.64	47	43	37	46	7	115.92	42	57	65	76	6.27	191.52	62	47	57	63	7.4	158.76
13	6_6	14	20	26	59	5.24	147.84	46	43	37	45	6.64	113.4	41	57	63	75	6.21	189	61	47	56	62	7.1	156.24
12	5_6	13	20	25	58	5.04	145.04	45	42	36	45	6.3	113.4	41	56	63	75	6.19	189	60	46	56	62	7.03	156.24
11	4_6	13	19	24	56	4.9	142.24	44	42	36	44	5.89	110.88	41	55	62	74	6.12	186.48	60	45	55	61	6.87	153.72
10	3_6	12	18	23	55	4.47	139.44	44	41	35	43	5.47	108.36	40	54	62	74	6.1	186.48	59	45	55	61	6.82	153.72
9	3_5	12	17	22	53	4.18	133.84	43	41	35	42	4.96	105.84	40	53	61	74	5.98	186.48	58	44	54	61	6.74	153.72
8	3_4	12	16	21	52	3.99	131.04	42	40	34	42	4.13	105.84	40	57	64	73	5.94	184.8	58	44	54	60	6.6	151.2

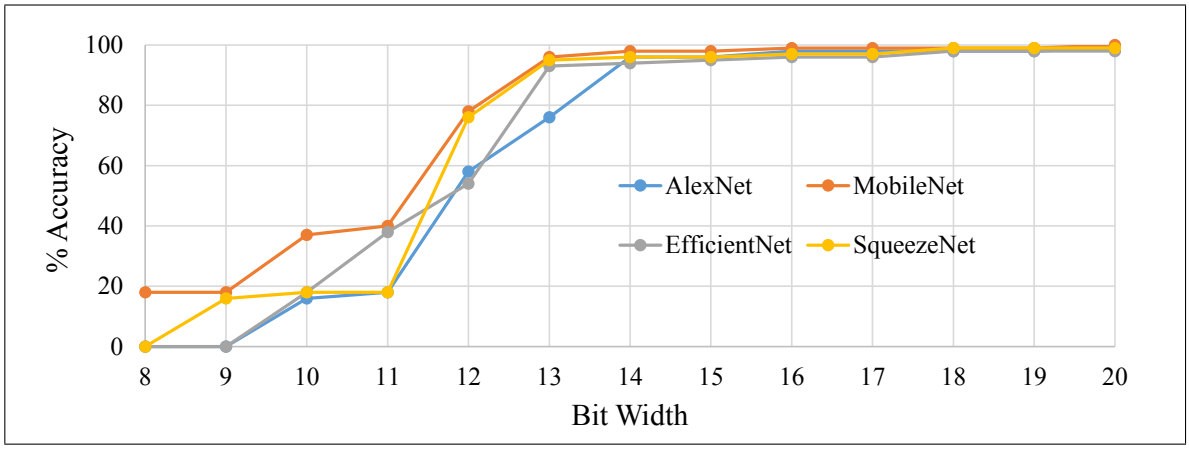


Figure 4.9: Top-1 accuracy for different bit width for different DNNs [30] [31] ©2021 IEEE.

of each of the four DNN models. Based on that, we reported the resource utilization, power consumption, and peak performance shown in Table 4.18. Further, top-1 and top-5 accuracy computation for different DNN models is shown in Fig. 4.9 and Fig. 4.10. For computing the top-1 and top-5 accuracies, we exploited the different orders of bit width on the last layer of the DNNs, where the softmax function is applied and the image class is derived. The accuracies obtained are relative to the performance of the original network. The overall FPGA resource utilization for the inference DNN models consumed 1253 (49%) BRAMs, 2E6 (50%)

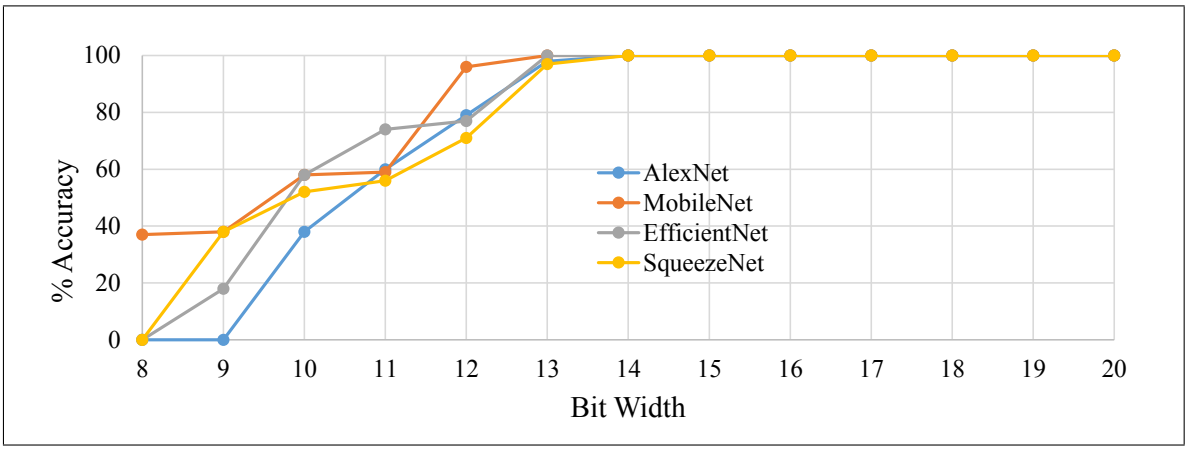


Figure 4.10: Top-5 accuracy for different bit width for different DNNs[30] [31] ©2021 IEEE.

Table 4.19: Layout results of the ASIC version [31] ©2021 IEEE.

Technology	GF 22nm FDSOI SLVT
Chip Size	1mm × 1.2mm
Core Area	$805\mu m \times 805\mu m$
Memory Type	SRAM (375KB)
Total Gate Count	785K
Frequency	800MHz
Precision	Fixed 16-bit
Total PE's	1024
Power	791mW
Latency	2.85ms
Performance	1.63 TOPS
Performance/W	2.06 TOPS/W
Energy	2255 μJ
DNN Models Inferenced	AlexNet, MobileNet, SqueezeNet and EfficientNet

FFs, 1E6 (55%) LUTs, and 2016 (70%) DSPs on the Virtex Ultrascale FPGA for the device XCVU440-flga2892-3-e.

4.4.5 Comparison with Previous Architecture

A comparison of the proposed accelerator design with prior SoA accelerator designs based on power, latency, performance, performance/watt, DSP, and DSP efficiency is shown in Table 4.20. Our work significantly surpasses other works, mainly in terms of power usage. The proposed accelerator consumes 2.5× less power compared to [87] and 4× lesser compared to [127] for AlexNet inference. Similarly, power consumption is less than 10W for other networks, whereas these networks are several MBs larger when compared to AlexNet and VGG-16. For the performance/watt comparison, proposed accelerator is 1.57× better compared to [127] and more than 3.63× better compared to other state of art designs [87] [118] [122] [128] [86] [127] [129]. Our accelerator offers a latency of 10ms and offers a high speed up for all the inference DNN models.

Table 4.20: Comparison with SoA FPGA accelerators. For our accelerator, we have benchmarked latency, performance, and DSP efficiency of AlexNet, MobileNet, SqueezeNet, and EfficientNet [30] [31]. ©2021 IEEE.

Parameters	**Zhang [87]**	**Qiu [118]**	**Zhang [122]**	**Suda [128]**	**Farabet [86]**	**Li [127]**	**Wu [129]**	**This Work**			
Device	Virtex 480t	Zynq xc 7Z45	Virtex 690t	Stratix V GSD8	Virtex6 ML605	Xilinx VC709	Xilinx XCZU9EG	Virtex Ultra Scale - XCVU440			
Precision	32-Float	16-Fixed	16-Fixed	16-Fixed	16-Fixed	16-Fixed	8-Fixed	16-Fixed			
Technology	65nm	28nm	40nm	28nm	40nm	28nm	16nm	20nm			
Freq (MHz)	100	150	150	120	200	156	333	180			
DNN Models	AlexNet	VGG 16	VGG16	VGG16	LabelMe	AlexNet	MobileNet	AlexNet	MobileNet	SqueezeNet	EfficientNet
Power	18.6	9.63	26	25.8	10	30.2	N/A	7.5	9.3	7.6	8.5
Latency (ms)	N/A	N/A	65.13	46.1	N/A	2.56	0.16	5.4	7.3	5.9	6.7
Performance (GOPS)	61.62	136.97	150	136.5	160	565.94	1378	223.69	165.47	204.73	180.29
Performance/w	3.31	14.22	5.77	5.29	16	18.74	N/A	29.82	17.79	26.93	26.9
DSP	2240	780	2833	246	N/A	2144	2070	2822			
DSP Efficiency (Performance/DSP)	0.03	0.18	0.05	0.55	N/A	0.26	0.67	0.079	0.058	0.0725	0.063

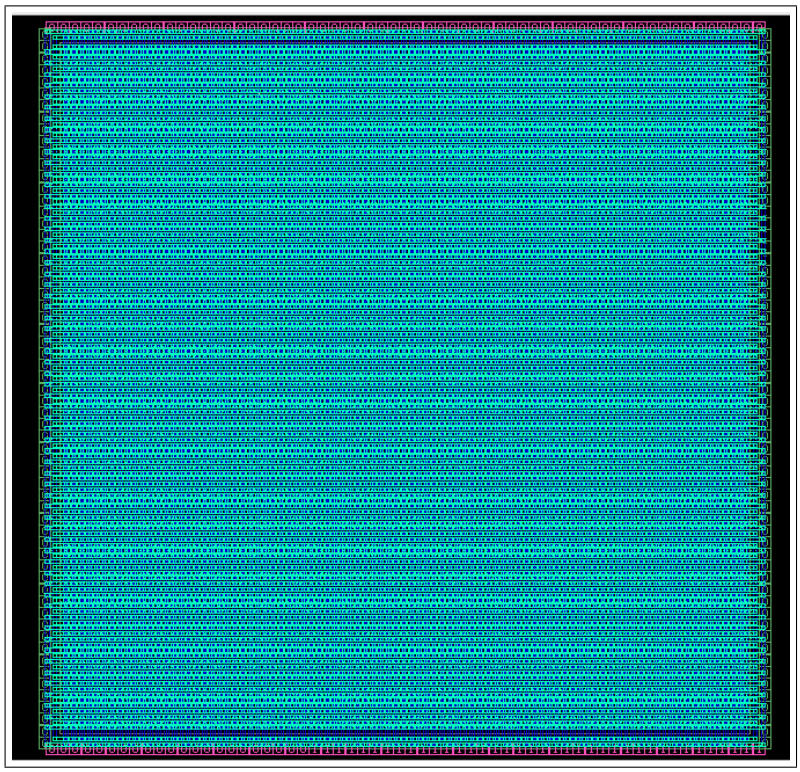


Figure 4.11: Layout view [31] ©2021 IEEE.

Table 4.21: Power and area breakdown of the ASIC version [31] ©2021 IEEE.

Modules	Area (μm^2)	Power (mW)
AXI Master	251.33	14.32
Control Unit	206.66	10.45
ReLU	13.97	2.53
PE_Array	446137.62	350.12
SRAM	200418.42	359.87
Bias Mem	146.10	7.36
Pooling	78.36	36.78
Truncator	28.36	10.53
Total	647252.48	791.96

Evaluating inference on different DNN models using various analytical studies shows that we can start with higher-order bit width and successively truncate the bits as we go down the layers. This way, we could save around 49% of power and an average of around 74% of resources due to efficient data movement and a robust truncation scheme while maintaining accuracy.

4.4.6 ASIC Results

We implemented the ASIC version of our accelerator in the 22nm CMOS process. Table 4.19 shows the layout specification, and the layout view is shown in Fig. 4.11. For the SLVT, we employed the SLVT process to meet the timing requirement of 800MHz, which was not possible in the LVT and RVT processes. The Table 4.19 was implemented for a fast corner at 0.88V

VDD at a temperature of 125°C. The ASIC version is for the multi-bit accelerator, as shown in Fig. 4.1, and it remains the same for all the DNN models inference. In the ASIC version, we could achieve a total throughput of 2.06 TOPS/W with a total power consumption of 791mW, a static power consumption of 317mW, and a dynamic power consumption of 474mW. The inference latency was around 2.85ms, and the energy consumption to achieve the total throughput was 2255μJ. The area and power breakdown are shown in Table 4.21, where the major area and power consumption is by the SRAM memory and the PE array.

4.5 Summary

Deep neural network models consume significant power, energy, and memory. Therefore, deploying them on a mobile platform is resource-expensive. To solve this drawback, we propose a multi-bit architecture. We start with a large bit-width of 32 bits, truncate the bits as we propagate through the network, and finally compute the last layer on a lower bit-width of 10 bits. An extensive memory-intensive network such as AlexNet, MobileNet, EfficientNet, and SqueezeNet was used as a benchmark model, and the architecture was evaluated on Virtex Ultrascale FPGA. While maintaining 100% top-1 accuracy, our proposed multi-bit architecture saves up to 49% of the power compared to its 32-bit fixed-bit architecture. Resources utilization was 74.60% for BRAMs, 68.76% for FFs, 73.25% for LUTs, and 79.425% for DSPs, less when compared with 32-bit architecture. The top-5 accuracy was maintained up to a reduced bit width of 10 bits at the last layer. The design achieves a peak performance of 130.3 GOPS/W on a Virtex Ultrascale FPGA. The design has an overall gain of 8.15× throughput compared to other prior FPGA accelerators. In addition, the overall power consumption is 4.5× lower when compared to other prior architectures. The ASIC version of the accelerator was designed in a 22nm FDSOI CMOS process to achieve an overall throughput of 2.03 TOPS/W at a power consumption of 791mW and a surface area of 1.2mm × 1.2mm.

In future work, the architecture can be extended to incorporate the support for the depthwise operation by providing separate channels for each channel without performing the final accumulation. In addition, for each channel, a separate ReLU, pooling, and truncator module would be used.

5

FantastIC4: A Hardware-Software Co-Design Approach for Efficiently Running 4bit-Compact Multi-layer Perceptrons

5.1 Introduction

In recent years, the topic of "edge" computing has gained significant attention due to the benefits of processing data directly at its collection source [130]. For instance, latency issues can be significantly mitigated by running machine learning algorithms directly at the edge device (e.g., wearable). In addition, increased privacy can be guaranteed since no data must be sent to third-party cloud providers. Their high predictive performance has triggered interest in deploying deep learning models to such embedded devices. However, traditional deep learning models are usually resource-hungry since they entail many parameters. In particular, processing many parameters requires expensive hardware components such as large memory units. Moreover, if high throughput and low latency are desired, many multipliers for parallel processing are high. This comes at the expense of spending many hardware resources on high power consumption and chip area, significantly limiting their application in use-cases with tight area and power consumption budgets, such as in the automotive, Internet of Things (IoT), or wearable.

This motivates the research of methods that can highly compress the DNN's weight parameters since, by doing so, we minimize the respective data movement and, therefore, its power consumption and the required chip area during execution. However, the efficient processing of compressed data representations comes with challenges, including bit-alignment problems, reduction of locality, and increased serialization. Moreover, SoA compression techniques require complex decoding before performing arithmetic operations, limiting the savings attained from compression, especially when the hardware is not tailored to such

decoding algorithms. This motivates a hardware-software co-design paradigm where, on the one hand, novel training techniques that make DNNs highly compressible are proposed. On the other hand, novel hardware architectures support the efficient, on-chip execution of compressed representations.

This work proposes a software-hardware optimization paradigm, which allows for the efficient execution of highly compact representations of DNNs based on FC layers. We specifically focus on fully-connected layers since they are usually the most significant size in a typical DNN model, and their execution is fundamentally more memory-bounded than other types of layers (e.g., convolutional layers). Moreover, many popular DNN architectures are entirely composed of FC layers, such as LSTMs and Transformers, which are highly relevant for time series and natural language processing tasks. Moreover, MLPs are already the status quo in use cases with very tight resource constraints since many studies identified MLPs to be one of the best algorithms to solve tasks in the IoT domain using wearable devices [131]. We apply several optimization techniques from both the hardware and software fronts, all tailored to increase the area efficiency and lower the power consumption of inference. We aim to make SoA MLP models more amenable, for example, in the applications above.

Our contributions can be summarized as follows:

- Firstly, we design a specialized hardware accelerator, named *FantastIC4*, which implements a first ACM technique to minimize the required number of multipliers for inference down to only 4 (thus the name of the architecture). By implementing ACM, we significantly reduce the computational resource utilization compared to the usual MAC paradigm, naturally due to performing less multiplication in total, but also due to better data movement of the activations for MLP models (activation stationary) as well as the reduction in the required area and power consumption for computations.

- FantastIC4 also supports the efficient, on-chip execution of multiple compressed representations of the weight parameters of FC layers. This boosts the compression rate of the layers, consequently improving the off- and on-chip data movement and reducing power consumption and area requirements since smaller-sized memory units can be implemented.

- To make the models amenable for the efficient execution on FantastIC4, a novel training algorithm is proposed that makes the models robust to 4-bit quantization while simultaneously encouraging low entropy statistics of the weights. Furthermore, explicitly enforcing low entropy statistics reduces the parameters' size requirements and simultaneously encourages sparsity, which is exploited by converting the parameters to compressed sparse formats.

- Our experimental results show that we can save 80% energy by compression and avoiding unwanted data movement between the (DDR3) DRAM and the on-chip SRAM and 75% of power by handling the 4-bit precision and sparsity in the processing elements (PEs).

- We evaluate the FantastIC4 on FC layers of popular DNN models and custom MLP trained on hand gesture and speech recognition tasks. We compare our accelerator to

other SoA FPGA and ASIC accelerators and see an improvement by 51× in terms of throughput and by 145× in terms of area efficiency (GOPS/mm^2).

In Section 5.2, we describe the other SoA techniques both on the hardware and software platforms. Section 5.3, we describe the need for using 4-bit quantization and how we handle the sparsity. Section 5.4 explains the training of the 4-bit-compact DNNs. The complete hardware architecture with PE design and other floating-point operations is described in Section 5.5. The experimental methodology is explained in Section 5.6, followed by the conclusion in Section 5.7. This chapter's content is published in the journal article [32], as the author of this thesis is also the first author of the published journal.

5.2 Related works

In the past few years, a great deal of work has been published on the efficient processing of DNNs, including topics such as neural architecture search, pruning and sparsification, quantization, compression, and designing specialized hardware architectures. An excellent overview of the landscape of different approaches and techniques studied on this topic is in [132] [133].

5.2.1 Techniques for reducing the information content of the DNNs parameters

The previous compression technique [54] pioneered a paradigm based on chaining sparsification, quantization, and lossless compression methods to reduce the redundancies in DNN's weight parameters significantly. As a result, [54] could compress SoA DNN models by up to 49×. However, several follow-up works have improved on all three fronts: compression, quantization, and sparsity.

Lossless Compression. Through a quantization scheme that minimizes the rate-distortion function while simultaneously considering the quantization to improve overall DNN performance, the work [51] demonstrated that compression gains on the same models as VGG-16 could be boosted to 63× by coupling quantization with a powerful universal entropy coder. However, although the proposed method achieves impressive compression gains, the resulting representation of the DNNs weights requires decoding to perform inference. In contrast, similar to the *Compressed Sparse Row* (CSR) matrix format employed in [54], [134] derives a representation that compresses the weights and enables inference in the compressed representation without requiring decoding. Furthermore, [134] showed that their proposed *Compressed Entropy Row* (CER) matrix format is up to 2× more compact and efficient than the CSR format when applied to DNNs. Lastly, compression offers a high advantage of reduced memory storage and faster inference time. However, there will be a slight disadvantage in loss of accuracy if the models are compressed properly.

Quantization. In recent years, researchers have been able to push the limits of quantization more and more. In particular, there is a growing amount of work showing that extreme quantization of the weights down to 4-bits is possible while minimally affecting the prediction accuracy of the network [135, 136, 137, 138, 139, 140, 141]. A 4-bit quantization offers directly 8× compression gains and similar improvements in computational efficiency. More robust

quantization techniques such as ternary and binary networks have also been proposed [142, 143]. Although they offer highly efficient implementations on a hardware level, they usually come at the expense of significant degradation of the accuracy of the network.

Simultaneous Optimization of Sparsity, Quantization and Compression. Some recent work has attempted to derive a unified framework for sparsifying, quantizing, and compressing DNN parameters. In particular, some have proposed novel regularizers that constrain the entropy of the weight parameters during training, thus explicitly minimizing the information content of the weights [144, 143]. Concretely, in these works, the first-order entropy is considered the entropy value as measured by the empirical probability mass distribution of the parameters. This regularization technique is theoretically well-motivated, directly measures the possible size reduction of the model, and simultaneously encourages sparsity and quantization of the weights too low bit-widths. These works were able to attain SoA compression results, e.g., [143] was able to train highly sparse and ternary DNNs, becoming one of the top 5 finalists in the NeurIPS19 Micronet Challenge[1].

5.2.2 Hardware Accelerators

There are many hardware accelerators from both academia and industry that concentrate on high performance and energy efficiency. Some of the topics that have been studied and analyzed are:

Data Flow Movement. Data flow movement is one of the important aspects of designing the hardware accelerators for any AI application. Activations and weight movements that are effective reduce energy requirements. The work in [90] provides an effective row stationary method and competent reusing of weights, Ifmaps, and PSums reuse. The DNN accelerator in [145] was manufactured using a 28nm process, supports parameter reuse, detects circuit timing violations to minimize worst-case guard bands and is tolerant of algorithmic noise. To maximize kernel efficiency and decrease off-chip memory access, the CNN-Recurrent Neural Network (RNN) processor in [146] implemented LUT-based multiplication and quantization table-based matrix multiplication. The PSums truncation from each of the preceding layers and performing inference on the truncated PSums and weights was shown in [30]. Bit Fusion [147] dynamically shared the weights across the different layers of a DNN model. The FantastIC4 reduces the data movement by 4-bit precision and uses FIFOs as a data buffer. Bitmask encoding is used to fetch the data from the FIFOs based on sparsity. In addition, FantastIC4 also supports the effective handling of layer weights by fetching the bitmask encoded non-zero values in a FIFO manner. Lastly, the floating-point operations are pipelined to save the dynamic power without compromising accuracy.

Systolic Arrays and Bit Serial Computations. The systolic arrays were first reported in [148] to efficiently compute DNN models to reduce unwanted power consumption when performing the MAC and matrix-vector multiplication. Using binary weights in [102] reduced the need for expensive multiplications. The bit-serial MAC operation was used in [149] to conserve the MAC operation's energy and save the chip's area. In FantastIC4, only four multiplications are performed with a proposed ACM paradigm.

[1] https://micronet-challenge.github.io

Sparse Data Compression. The compression with sparsity and pruning was shown in [54] to fit the DNN models and on-chip SRAM. Based on the pruning and sparsity, the hardware accelerator is implemented in [150], and it is 19× more energy efficient than the uncompressed versions [28] [31] [62]. The compression was extended to convolutional layers in [151]. The weights and activations were compressed using CSC format [152]. The scalpel accelerator [153] showed that the weight pruning achieves a total speedup of 1.9×. In contrast to FantastIC4, all mentioned accelerators support only one compressed format, significantly limiting the attainable compression gains and, consequently, the power savings from off-chip to on-chip data movement.

Analog and Mixed Signal Accelerators. Even though the digital DNN accelerators are in high prominence, analog and mixed-signal accelerators get the attraction to the AI hardware domain. The main goal of the [154] is to implement the minimum energy requirement for CIFAR-10 using binary networks. Here, the XNOR operation replaces the multiplication operation, as the weights are constrained to +1 and -1. This technique allows off-chip memory access to be avoided. The mixed-signal hardware error handling was reported in [155] to maintain good accuracy and have a low bit precision of 2 bits. Even though analog and mixed-signal hardware are less complicated in the implementation, the robustness in handling large data sets or maintaining the accuracy could be better. FantastIC4 uses non-binary weights and activations but maintains higher accuracy, better throughput, and low power consumption, as explained in other sections.

FPGA based Accelerators. Several FPGA accelerators have proposed solutions for optimized accelerator designs in the industry and academia. The energy-efficient FPGA accelerator [156] performed inference on CNN with binary weights. The processor achieves a throughput of 2100 GOPs with a latency of 4.6ms and power of 28W. The hardware-software co-design library to efficiently accelerate the entire CNN and FCN on FPGAs was shown in [122]. The floating-point arithmetic CNN accelerator [157] introduced an optimized quantization scheme based on rounding and shifting operations. They reported an overall throughput of 760.83 GOPs. The other accelerators worked on sparse matrix-vector multiplications mainly for the MLPs [158] [159]. Even though these accelerators perform well, they still lack throughput, power, or latency requirements. The FantastIC4 FPGA version utilizes an efficient computation approach to achieve high throughput with minimal power, latency, and resource requirements.

5.3 FantastIC4 Design

In this work, we propose to apply several optimization techniques that, in combination, are tailored to reduce the area and energy requirements for performing inference. The main idea is to minimize the memory requirements and the number of multiplications needed to perform inference since both are the primary sources of area utilization and power consumption.

5.3.1 4-bit quantization

As mentioned in the related work Section 5.2, it is well known that quantization is a powerful technique for lowering the memory and computational resources for inference [132, 133]. However, the increasing demand for deployment of DNNs on edge devices with very tight

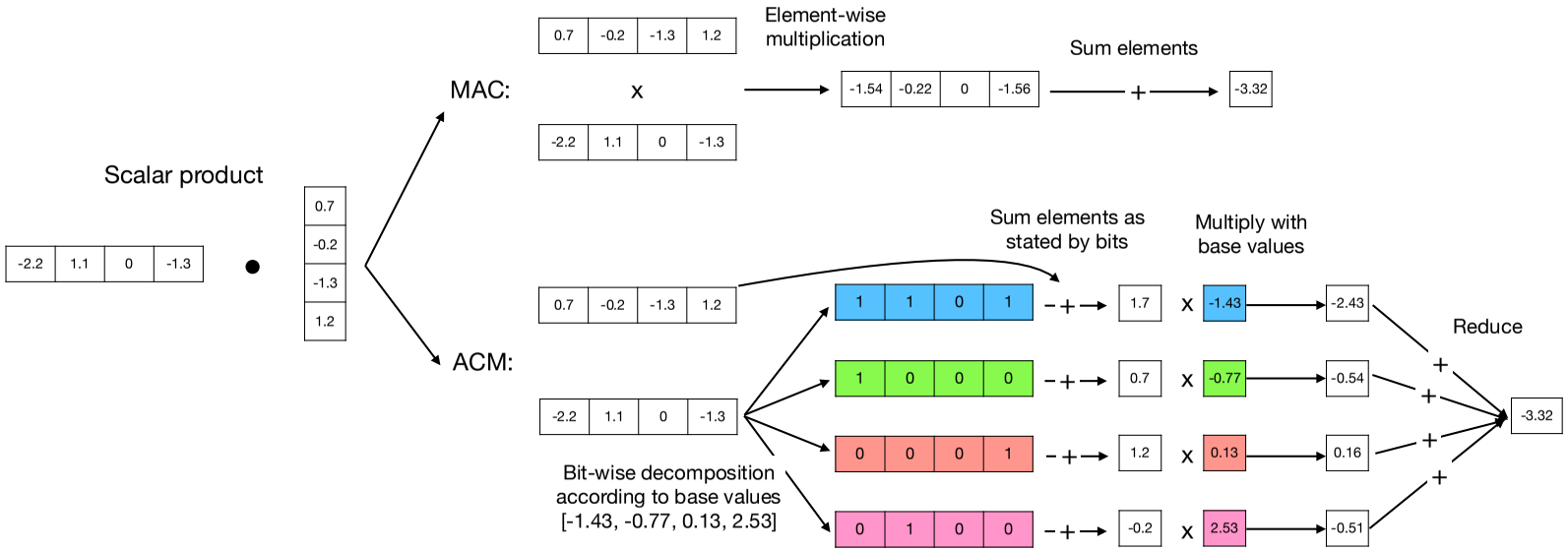


Figure 5.1: Sketch example on the different computational paradigms when performing the dot product algorithm [32] ©2021 IEEE.

hardware constraints (ex: microcontrollers) has pushed researchers to investigate methods for extreme quantization, resulting in weights with merely 4 bits or lower. The result is an 8× compression of the model, which helps minimize the cost of the off and on-chip movement of weights data. In particular, FC layers have shown to be highly redundant and robust to extreme quantizations down to 4-bit [54, 150], again, our work's primary focus.

5.3.1.1 Increasing the Computational Efficiency

However, the inference modules of extremely quantized layers are often implemented following the usual multiply-accumulate (MAC) computational paradigm. We argue that this computational paradigm could be more efficient in the regime of extremely low precision. Instead, we propose first accumulating the activations at each bit level and multiplying the results, thus creating an accumulate-multiply (ACM) computational paradigm. ACM is a unique form of Distributed Arithmetic (DA) used in DSPs for dot product computation between a fixed and a variable data vector. In [160], DA was used for the first time in a deep-learning scenario. More concretely, we follow the equation described below.

$$\underbrace{W \cdot A}_{\text{MAC}} = \left(\sum_{i=0}^{3} \omega_i B_i\right) \cdot A = \underbrace{\sum_{i=0}^{3} \omega_i (B_i \cdot A)}_{\text{ACM}} \tag{5.1}$$

where we denote as W the weight parameters of, e.g., a fully-connected layer, A the input activations, $\cdot$ the operator denoting the dot product, and B_i a binary mask corresponding to the base ω_i. Thus, as shown in equation (5.1), we represent the weight parameters W as a linear combination of four binary masks B_i with respective coefficients ω_i. This representation generalizes any 4-bit representation that is applied to the weights. For instance, if $\omega_i = 2^i$, the elements of W are represented in the uint4 format. The uint4 format is a 4-bit unsigned integer format.

As one can see from the right-hand side of Eq. (5.1), we can first accumulate the activation values that are positioned as indicated by the bitmasks B_i, and then multiply the output by the base value (or base centroid) ω_i. As a result, the number of multiplications is significantly reduced. In this work's setup, only 4 multiplications are required per output element, which is almost negligible for large dimensions of input activations. For example, by utilizing a fully connected layer to map 1000 input activations to 100 output activations, the MAC paradigm requires 100,000 multiplications. On the other hand, the proposed ACM approach scales only with the number of output activations and requires 400 multiplications. Thus, the inference procedure is now dominated by the complexity of performing additions. Figure 5.1 sketches this procedure for a concrete example and compares it to the traditional MAC paradigm. Here, given two input vectors, the MAC calculates the respective scalar product by multiplying the elements and adding them. In contrast, the accumulate-multiply (ACM) first sums the elements of one of the vectors (in this diagram, the right-hand-side vector) according to the bit-decomposition of the other, then multiplies the respective basis values and reduces the output. In the above sketch, the base values were [-1.43, -0.77, 0.13, 2.53], and we color-coded according to [blue, green, red, pink]. Thus, the original element values result by performing the linear combination in the vertical direction, for instance, $-2.2 = 1 \times (-1.43) + 1 \times (-0.77) + 0 \times (0.13) + 0 \times (2.53)$.

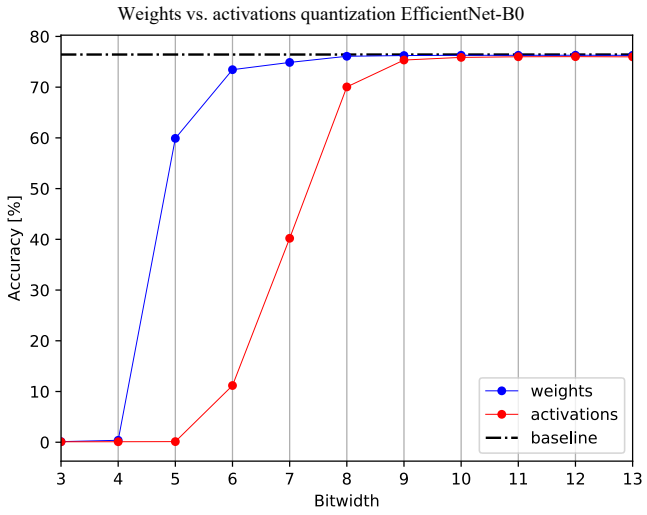


Figure 5.2: Difference in sensitivity between the activations and weight parameters of the EfficientNet-B0 model. Activations are more sensitive to quantization since the model's prediction performance drops significantly faster (at higher precision values) [32] ©2021 IEEE.

5.3.1.2 Increasing the Capacity of the Model

Moreover, the usual MAC computational paradigm requires quantizing the activations of the model down to 4 bits (or lower) to exploit the benefits of extreme quantization. Since activations are often more sensitive to perturbations than the weights, as shown in Fig. 5.2, this significantly degrades the neural network prediction performance. Moreover, bias and batch-normalization tend to be more sensitive than the weight parameters. Therefore, it motivates the support of mixed-precision layers, where input and output activations, bias, and batch-norm parameters can be represented with higher precision than weights to compensate for the degradation inaccuracy. FantastIC4 design supports higher precision activation values since this can be easily integrated within the ACM computational flow. In addition, we support the full-precision representation of the batch-norm parameters and the bias coefficients since their memory and compute costs are relatively low compared to the operations involved in the weight parameters.

In addition, in our work, we do not constrain the values of the linear coefficient ω_i to be of powers of 2, as it is most common in the MAC approach, but allow $\omega_i \in \mathbb{R}$. The expressive power of W and the model's capacity increases, allowing it to learn more complex tasks.

5.3.2 Why do we focus on low entropy?

Lowing the entropy of the weights has several benefits in terms of memory and computational complexity, as discussed in [134] in detail. We stress that by entropy, we mean the first-order entropy, as measured by the empirical probability mass distribution of the parameters. Concretely, $H = -\sum_i P_i \log_2 P_i$, where P_i measures the empirical probability mass distribution of the i-th cluster center. Additionally, low entropy is calculated from the DNN model, where we calculate the total number of times the weights have appeared in the whole model. The calculation is straightforward, where the probability of an element is equal to the number of times it appears in the DNN model divided over the total number of weights of the DNN. The probability mass distribution is calculated according to Eq.5.2, where E refers to the total

number of elements and W refers to the total number of weights. In the following, we explain how we leverage the low-entropy statistics of the weights in this work.

$$p_i = \frac{E}{W} \tag{5.2}$$

5.3.2.1 Saving Arithmetic Operations

Low entropy statistics encourage sparsity [134]. As thoroughly explained in previous work [150, 90, 151], sparsity allows to save computations by skipping zero-valued operations. In particular, FantastIC4 does not perform additions of activations when zero-valued weights are present, thus saving on arithmetic operations and, consequently, dynamic energy consumption.

Moreover, low entropy statistics also encourage the low number of unique non-zero values, thus a high probability of encountering the same non-zero value. This property can be exploited when loading non-zero values by reducing the dynamic power required when loading the same value. A simple example of how the arithmetic operations are saved by employing the accumulate-multiply technique instead of the MAC technique is shown in Fig. 5.1.

5.3.2.2 Multiple Lossless Compression

There are several ways to compress sparse weights. One is by converting the weights in the *Compressed Sparse Row* (CSR) format [54], which is based on applying run-length coding for saving the signaling of the positions of non-zero values. Another is using a simple form of Huffman coding, which consists of storing a bitmask indicating the positions of the non-zero values followed by an array of non-zero values organized in, e.g., row-major order. In the high sparsity regime (>90% of zeros), the CSR format attains higher compression gains, whereas, for smaller sparsity ratios (25% - 90% of zeros), the Huffman code compresses the weights. Since the sparsity ratio of different layers can vary significantly, FantastIC4 supports processing both sparse representations on-chip. Furthermore, the bye model can increase compression gains and reduce transmission costs from off to on-chip by enabling more flexible compression opportunities.

5.4 Training of 4-bit compact DNNs

As described in the previous Sections 5.3, our proposed optimization paradigm is based on the fact that the weight parameters exhibit low-entropy statistics and can be represented with 4-bits. However, suppose we naively lower the entropy and strongly quantize a pre-trained model. In that case, we will most likely incur a significant drop in accuracy, as explained in the experimental section 5.6. Therefore, we propose a novel training algorithm in this work that makes DNN models robust to such transformations.

5.4.1 Entropy-constrained training of DNNs

Our method is strongly based on EC2T, proposed in [143] that trains sparse and ternary DNNs to SoA accuracies. Instead, we generalize their approach so that DNNs with 4-bit weights

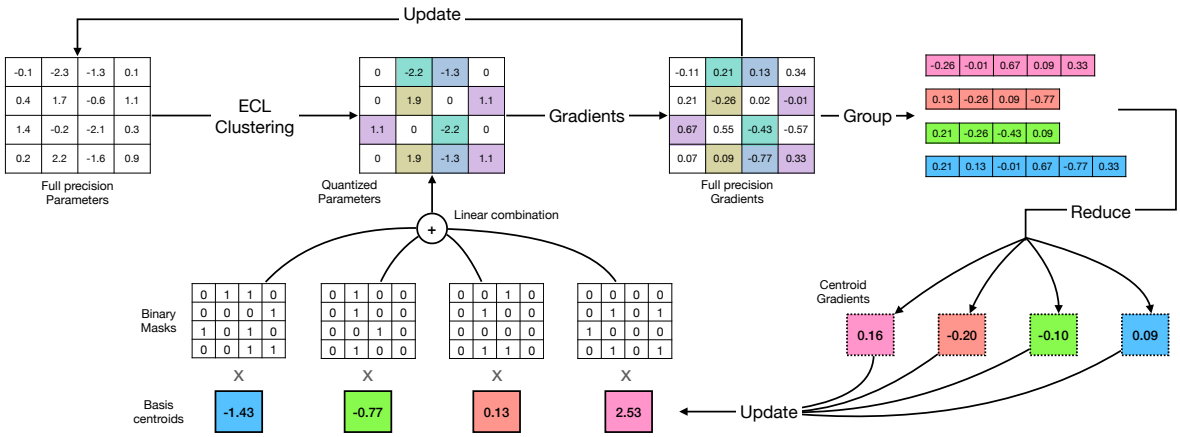


Figure 5.3: Entropy constraining of DNNs [32] ©2021 IEEE.

and low entropy statistics are attained. Concretely, our training algorithm is composed of the following steps:

1. Quantize the weight parameters (but keep a copy of the full-precision weights) by applying the entropy-constrained Lloyd (ECL) algorithm [161].

2. Apply the straight-through estimator (STE) [162] and forward + backward pass the quantized version of the model.

3. Update the full-precision weights and the centroids with the computed gradients.

Fig. 5.3 sketches the training method. The 4-bit entropy-constrained training method for compressing DNNs is based on the straight-through estimator (STE). Firstly, the full-precision parameters are quantized using the entropy-constrained Lloyd (ECL) algorithm, whereas the quantization points are constrained to be linear combinations of 4 bitmasks with 4 basis centroids. Then, the gradients are calculated concerning the quantized DNN model. The full-precision parameters are updated respectively, whereas each basis centroid's gradients are computed by grouping and reducing their respective shared gradient values.

5.4.2 Definition of the centroids

As described in equation (5.1) (section 5.3), we represent the weight parameters W of the DNN as a linear combination of 4 binary masks B_i with respective coefficients ω_i. With this, we can define 16 different cluster center values (or centroids), four of which are the coefficients ω_i and the rest a linear combination. To increase the capacity of the models, we assign to each weight parameter W his unique set of four centroids Ω.

5.4.3 Entropy-Constrained Lloyd algorithm (ECL)

The ECL algorithm is a clustering algorithm that also considers the entropy of the weight distributions. Throughout this work, we define entropy as $H = -\sum_i P_i \log_2 P_i$, where P_i measures the empirical probability mass distribution of the i-th cluster center. To recall, the H states the minimum average amount of bits required to store the output samples of the distribution [163]. Thus, ECL tries not only to minimize the distance between the centroids and

the parameter values but also the information content of the clusters. Again, this regularization term is theoretically well-motivated, directly measures the possible size reduction of the model, and encourages sparsity + quantization of the weights to low bit-widths.

However, we slightly modified the algorithm so that the ECL method does not update the cluster centers. Instead, we fine-tune the cluster centers with the information from the gradients (more in subsection 5.4.5).

5.4.4 Making DNNs robust to post-training quantization

As stated earlier, the accuracy drop may be significant if we naively apply the ECL algorithm to a pre-trained network. Therefore, we use the STE method [162] to make them robust to extreme quantization. In the case of neural networks, this means applying further training iterations to update the full-precision parameters about the gradients computed by the quantized parameters. We adapt the full-precision weight parameters to the prediction error incurred by the quantization, thus forcing them to move to minima, where they are robust to ECL-based quantization.

5.4.5 Fine-tuning centroids

Our particular contribution is reflected in the definition of the 16 clusters and their respective gradient propagation (i.e. fine-tuning). To recall, we represent each (quantized) weight parameter as a linear combination of 4 binary masks B_i with respective coefficients ω_i, thus $W = \sum_{i=0}^{3} \omega_i B_i$. Therefore, we only update the four basis centroids ω_i at each training iteration since 12 out of the 16 centroids are linear combinations of these. Hence, we calculate the gradients δ_i^{ω} of each centroid ω_i as follows: Let δ^W be the gradient tensor of the weight parameter W, then

$$\delta_i^{\omega} = \sum_{j=0} \delta_j^W B_i \tag{5.3}$$

with B_i being the binary mask respective to the coefficient ω_i, and j being the dimension that iterates over all parameter elements. After computing the gradient of each centroid, we update them by applying the ADAM optimizer. The simple example of fine-tuning of centroids is shown in 5.1.

5.5 Hardware Architectures for FantastIC4

Fig. 5.4 shows the overview of the FantastIC4 system. The entire system is a heterogeneous combination of a CPU and FPGA. The system consists of the CPU, the external DDR3 memory, and the FPGA chip. The software part mainly consists of the CPU that transfers the input data and the DNN model (only one time) to the FPGA. Since all the data is usually extensive and can not entirely be stored on an on-chip BRAM, some of it is stored in an off-chip DRAM. The data is then accessed through a memory controller built across a Memory Interface Generator (MIG) IP. We have the FantastIC4 control unit, memory controller, I/O Buffers, and the FantastIC4 accelerator on the FPGA chip. The memory controller facilitates the movement of the input data from off-chip DRAM to the accelerator and stores the computation results into the DRAM. The control unit controls the behavior of other modules on the FPGA.

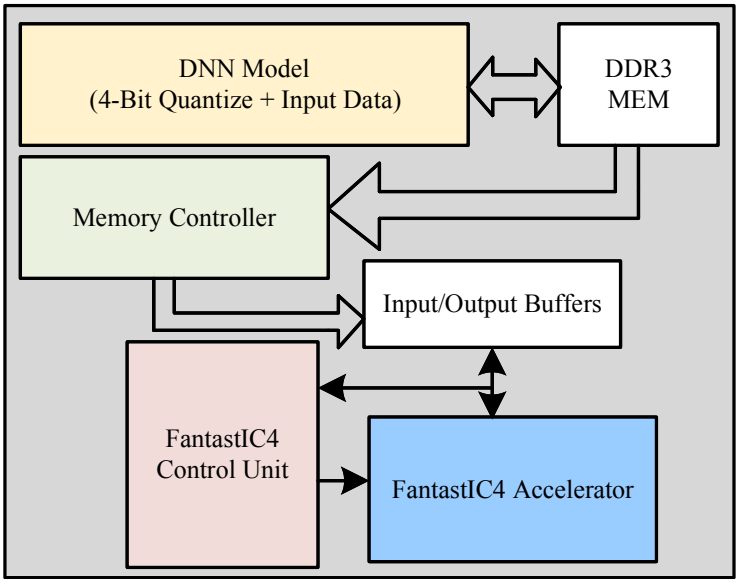


Figure 5.4: FantastIC4 system [32] ©2021 IEEE.

Table 5.1: Control states of the FantastIC4 control unit [32] ©2021 IEEE.

Data Movement	-	Acts,Wt,Bias alpha and CSR	CSR Data	-	-	-	-	-	-	-
Computation	-	-	BM Conv	Wt ID	Add tree/ MAC	Fix-Flt	Flt Mul1	Flt Add	Flt Mul2	Flt-Int
State	Start	State1	State2	State3	State4	State5	State6	State7	State8	State9
Time(ns)	0	5000	10	10	10	30	50	50	40	20

It is responsible for data movement and computation inside the accelerator. The I/O buffers store the input data for processing and return the PSums data from the accelerator for the subsequent layer inference. The FantastIC4 accelerator is the heart of the entire system, which reads the data from the DRAM, performs the computation, and stores the results back into the DRAM memory.

5.5.1 Memory Controller and Input/Output Buffers

The FantastIC4 accelerator accesses the DDR3 memory through a MIG interface operating at a clock frequency 200MHz. We employ the AXI communication protocol for the data movement between the FPGA chip and the off-chip DRAM. The MicroBlaze CPU and other AXI control IPs communicate through the MIG interface with the DDR3. The memory controller receives instructions from the FantastIC4 control unit through the AXI master to read and write the data from/to the memory. The I/O buffers provide dual buffering for the data movement in a ping-pong manner.

Our proposed accelerator has two levels of the control hierarchy. Table 5.1 shows the control states for our accelerator. The first level of the hierarchy, i.e. the Start and then State1, controls the data movement between the DRAM, memory controller, and the accelerator on the FPGA chip. Here, the activations, weights, biases, alpha values for floating-point operations, FIFO data, and 256-bit CSR Pointer data are moved into their respective memory/registers for computation. At this level, all the data movement operations are performed sequentially.

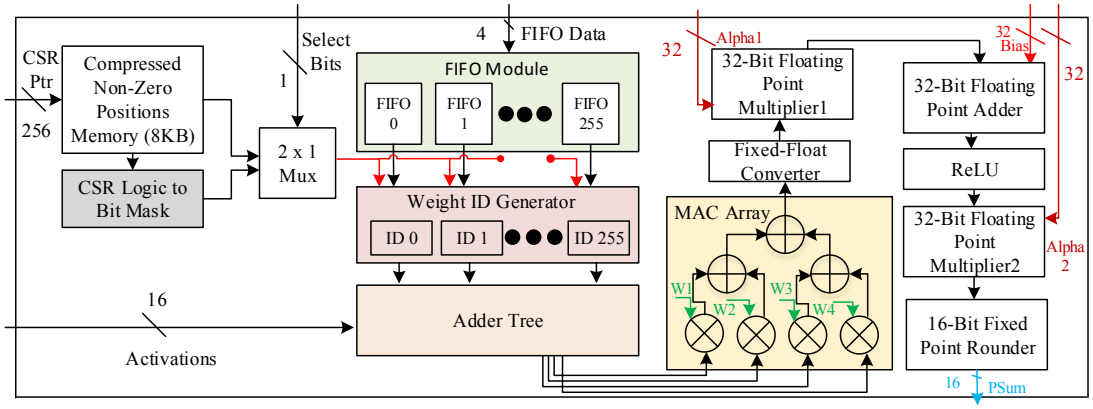


Figure 5.5: FantastIC4 architecture [32] ©2021 IEEE.

The total time to complete these two states is around 5000ns for MLP models. Here, the total time taken mainly depends on the DNN model under inference. At the next level of the hierarchy, we perform the computations. State2-State9 shows the various stages of processing performed on the accelerator. The different orders of computation performed are CSR to bitmask conversion, weight ID generation, accumulation and multiply operation, and finally, the single-precision floating-point operation. The total time taken to perform the computation is around 220ns. The computation time is less because all the states work concurrently, and each state is independent of the other states except on the first iteration.

5.5.2 FantastIC4 Architecture

The top-level hierarchy of the FantastIC4 architecture is shown in Fig. 5.5. The architecture operates on a single clock frequency domain of 150MHz (FPGA-based) and 800MHz (ASIC-based). FantastIC4 is composed of CSR to bitmask logic to perform CSR to bit mask conversion, FIFO modules to store the weight IDs for 256 adder trees, and a weight ID generator fetches the data from the FIFO modules based on the outcome of CSR to bit mask conversion. An adder tree accumulates the activations based on the weights IDs from the ID generator. The MAC array performs four multiplication and three addition operations. The fixed point to floating point converter converts a 16-bit fixed point MAC output into a 32-bit single-precision float output. This 32-bit floating-point MAC output will be multiplied by a 32-bit alpha1 value; where the alpha1 value is an array of single-precision floating-point data, the output of the multiplier1 will be added with the bias. The output of the adder will undergo a non-linear activation operation called ReLU to perform the computation as explained in Eq. 2.3. Final floating-point multiplication is performed with another 32-bit single-precision alpha2 value. The 32-bit result from the multiplication will be rounded back to the 16-bit integer value to generate the final PSums.

CSR to Bitmask Logic. By default, FantastIC4 loads the positions of the non-zero elements of a row of the sparse weight matrix according to the compressed Huffman representation, which consists of a straightforward binary mask of width 256. The Huffman encoding algorithm is a data compression algorithm where the variable-length code is assigned

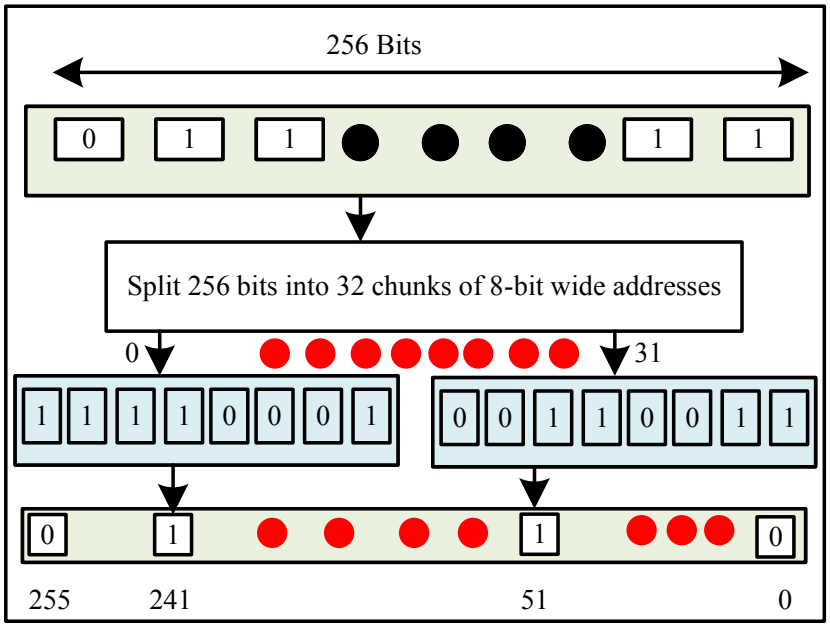


Figure 5.6: CSR to bitmask conversion logic [32] ©2021 IEEE.

to each input character, where the frequently occurring character gets the smallest code, and the least occurring characters get the largest code [164]. The bitmask controls the weight ID movement into the adder tree. However, when a layer of non-zero positions is compressed following the CSR format, a logic must be implemented that converts them back to a bitmask representation, which is the purpose of the CSR to bitmask logic. The conversion logic is shown in Fig. 5.6. The compressed non-zero position data pointers comprising 256 bits will be split into chunks of 32, which is 8-bit wide. Based on the 8-bit value, each bit of the encoded bitmask will be set to '1'. For example, As shown in Fig. 5.6, the 0^{th} chunk had a value of 241, and the 31^{st} chunk had a value of 51. So, the corresponding 241^{st} bit and 51^{st} bit will be set to 1, and the remaining bits will be set to 0 to generate a 256-bit encoded bitmask data. The CSR pointer and bitmask data will be selected from a 2×1 multiplexer through a "Select Bits" to generate the final encoded data for weight ID generation.

FIFO Module and Weight ID Generator. The FIFO module has 256 individual FIFOs with a width of 4 and a depth of 256, each storing the non-zero weight elements of a particular column of the layer's weight matrix. These FIFO modules are stored in an array of registers. The weight ID generator has a simple selection logic, where each IDs of 4 bits is fetched from the FIFOs based on the encoded bitmask data. For example, if the encoded bitmask is '1', an ID will be fetched from the FIFOs, and the pointer points to the exact location of the fetched data. The weight ID generator has 256 ID modules, which store the 4-bit IDs from the FIFO if the 256 individual bits from the bitmask are '1' or store 4-bit zero data.

Adder Tree and MAC Array. Adder trees consist of an array of 256 adders arranged in a logarithmic fashion. The adder tree has two stages: Adder Stage1 and Adder Stage2. Stage1 has 128 adders arranged in one group. Each adder in the stage1 has three hierarchy levels

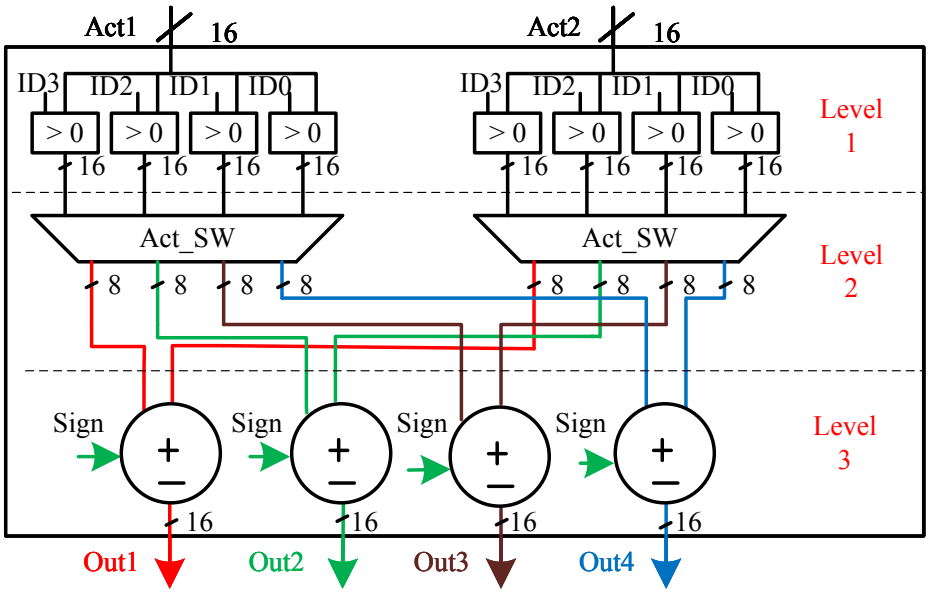


Figure 5.7: Adder schematic [32] ©2021 IEEE.

with a control parameter in each hierarchy. Fig. 5.7 shows the adder schematic in Stage1. The adders are fed with the weight ID generator's two different activations and two different IDs. All the activations in the adder tree are static and used for all computations cycles. The static activations in the adder tree enable significant power consumption. Having static activations inside the adder rather than accessing it from memory saves up to 15% of power consumption. In the level1 hierarchy, the 4-bit weight IDs control the movement of the activations inside the adder. Each bit from the weight IDs forms a channel that regulates the activation flow to level2. If the ID is 1, the 16-bit activation is fed, or a zero value is fed to the level2. Eight groups of activation data will come out of the level1 hierarchy. In the level2 hierarchy, the activation switches between the upper and lower half of 8 bits. This technique fits the more extensive networks into the hardware and improvises the prediction. In this hierarchy, if the activation switch is low, the lower or upper half of the bits is selected. In the level3 hierarchy, the actual computation is performed. The sign mode determines whether the activations need to be added or subtracted. Finally, four different computations are performed among the eight groups to generate the four-output data of 16 bits from each adder. The adder stage2 has 128 adders arranged in multiple groups. The first group in the adder stage2 has 64 adders, the second group has 32 adders, and similarly, other groups are logarithmically scaled down. The adders in the adder stage2 perform only the computation, unlike those in the stage1. Depending on the sign-bit in the output data from the stage1, either addition or subtraction is performed.

The MAC array performs four multiplications and three additions, respectively. Therefore, the four outputs, each of 16 bits from the adder tree, will be multiplied with the 16-bit basis weights to generate a 32-bit product, which we will accumulate to create the final 32-bit MAC output.

Floating Point Operations. The floating-point operation mainly comprises fixed to floating-point conversion, floating-point multiplications, floating-point addition, and final 32-bit

Algorithm 1 Fixed Point to Floating Point Conversion

```
mac_out: 32-bit fixed point MAC output
convert_out: Single precision floating point number
procedure FixedtoFloat(mac_out, convert_out)
    lod ← Leading one detector
    convert_out_sign ← Sign bit of floating point
    convert_out_exponent ← Exponent of floating point
    convert_out_mantissa ← Mantissa of floating point
    for k ← 30 downto 0 do
        if (mac_out[k] == 1) then
            lod = k;
        end if
    end for
    convert_out_sign = mac_out[31];
    convert_out_exponent = lod + 127;
    convert_out_mantissa = mac_out << (23 - lod);
    Combine sign, exponent, and mantissa to generate convert_out
    return convert_out
end procedure
```

floating-point to 16-bit integer conversion. In the fixed-to-floating-point conversion, a 32-bit fixed-point MAC data is converted into equivalent single-precision floating-point data as shown in the Algorithm. 1, a leading one will be detected from the MAC output, and a corresponding conversion operation is performed.

The converted floating-point data will undergo a single-precision floating-point multiplication with Alpha1 values. The Alpha1 values are stored in a SRAM of only 1KB. With these scaling factors, FantastIC4 can accommodate for de-quantization and batch norm parameters, as explained in Section 2.6.1.3. As shown in Fig. 5.8, both the inputs will be normalized and split into their equivalent sign, mantissa, and an exponent part. The 23-bit mantissa will be multiplied with each other to generate 48-bit output. The MSB of the multiplied output will be used to calculate the final mantissa and the exponent part. The sign bits of both inputs will be XORed to generate the final sign bit. The final sign, exponent, and mantissa will be concatenated to generate the final 32-bit multiplied output. Subsequently, the multiplied floating point will be added with the bias data stored in another 1KB SRAM. This operation is similar to the multiplication operation in terms of the normalization of the data. Then, the added data will undergo a non-linear activation ReLU function as $f(x) = max(0, x)$ since it is the status quo non-linear function for most MLP models. The example for this ReLU function is shown in Section 2.1. The ReLUed output will be further multiplied with a single 32-bit Alpha2 value to generate the final 32-bit multiplied output. These scaling factors consider different quantization parameters, which is vital for correctly calibrating the subsequent quantization step, consisting of a final rounding of 32 bits to a 16-bit integer. The 16-bit integer is the final PSum that will be used as an activation for the inference of the next layer.

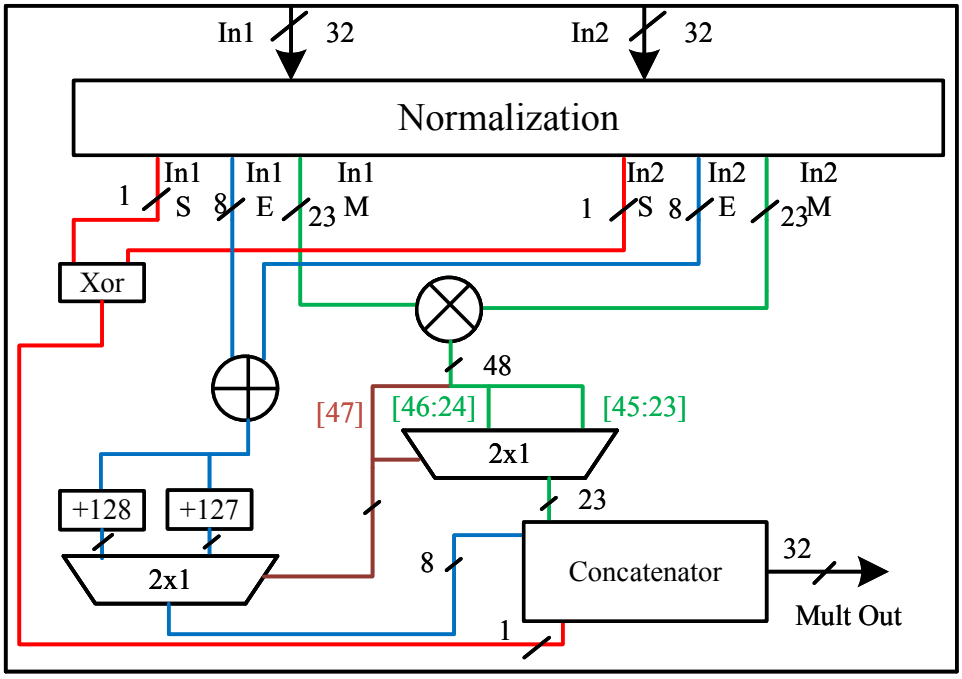


Figure 5.8: Floating point multiplier [32] ©2021 IEEE.

5.6 Experiments

5.6.1 Experimental setup

5.6.1.1 Datasets and Models

In the experiments section, we distinguish between hardware-conform and non-conform models. Those confirmed models that are fully compatible with our hardware architecture and a bit accurate can be used for the entire inference process. Consequently, conform models include only FC layers with 512 input/output features. Optionally, batch norm layers (as explained in 2.6.1.3) are allowed, which can result in accuracy gains.

To cover a variety of use cases with the conform models, we trained and deployed several models solving classification tasks for audio, image, biomedical, and sensor data. Concretely, we considered the Hand Gesture Recognition (HR) task based on the biomedical and sensor dataset, the Google Speech Commands (GSC) dataset for audio classification, and the MNIST and CIFAR-10 datasets for the small-scale image classification task. We trained and implemented custom and well-known MLPs for solving the tasks mentioned above. In addition, to benchmark our quantization algorithm, we also used non-conform models, which we have not trained ourselves but obtained from publicly available sources. Concretely, ResNet-50 and -34 come from the torchvision model zoo[2], EfficientNet-B0 [3] and ResNet-20 [4]. We further trained these models by applying our entropy-constrained method (Section 5.4) and benchmarked

[2] https://pytorch.org/docs/stable/torchvision/models.html

[3] https://github.com/lukemelas/EfficientNet-PyTorch, Apache License, Version 2.0 - Copyright (c) 2019 Luke Melas-Kyriazi

[4] https://github.com/akamaster/pytorch_resnet_cifar10, Yerlan Idelbayev's ResNet implementation for CIFAR10/CIFAR100 in PyTorch

their accuracies at different regularization strengths. We refer to the appendix for a more in-depth description of the experimental setup, the models, and the datasets employed in the experimental section.

Hand Gesture Recognition (HR). The authors in [20] collected Inertial Measurement Unit (IMU) and electromyogram (EMG) readings from 5 different subjects in 5 different sessions to capture 12 defined hand gestures. Different from [20], we deploy a small MLP to solve the classification task. It outperforms the proposed Hidden Markov Model, which achieves a mean accuracy of 74.3% for person-independent hand gesture recognition. Our proposed 4-layer deep MLP achieves a person-independent mean accuracy of 84.0%. Quantizing all network layers to 4-bit with the FantastIC4 algorithm is possible with almost no drop in accuracy. The model consists of an input layer, two hidden layers, and an output layer with 512, 256, 128, and 12 output features, where a BatchNorm layer follows each FC layer. The data corpus [5] is used for Hand Gesture Recognition.

Google Speech Commands (GSC). The Google Speech Commands dataset consists of 105,829 utterances of 35 words recorded from 2,618 speakers. The standard is to discriminate ten words "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", and "Go", and adding two additional labels, one for “Unknown Words”, and another for “Silence” (no speech detected) [19]. No overlapping speakers exist between the train, test, and validation sets. We deploy an MLP consisting of an input layer, five hidden layers, and an output layer featuring 512, 512, 256, 256, 128, 128, and 12 output features. Our model achieves a classification accuracy of 91.0%, outperforming the default CNN model (88.2%) in the TensorFlow example code mentioned in [19]. The FantastIC4 4-bit quantization regularly affects the full-precision MLP and further improves the classification accuracy to 91.35% while introducing 60% sparsity. The authors in [165] show that CNNs, especially RNNs, usually achieve better accuracy than MLPs for the GSC dataset. Still, our proposed model yields a comparable accuracy to their proposed CNN and outperforms their 8-bit quantized MLP (88.91%). For another comparison, [166] quantized their network composed of three convolution layers and two fully-connected layers to 7-bit using 8-bit activations and achieved an accuracy of 90.82%.

Image Classification. We utilized two neural networks for small-scale image classification. One MLP would fit our proposed accelerator, LeNet-300-100, and one CNN (ResNet-20). CIFAR-10 [167] is a dataset consisting of natural images with a 32×32 pixels resolution. It contains 10 classes. The train and test sets contain 50,000 and 10,000 images. MNIST [168] is drawn from 10 classes where each class refers to a handwritten digit (0-9). The dataset contains 60,000 training and 10,000 test images with a 28×28 pixels resolution. To benchmark our quantization algorithm with ImageNet, we deployed EfficientNet-B0, ResNet-50, and -34 networks. The ImageNet [169] dataset is a large-scale dataset containing 1.2 million training images and 50,000 test images of 1000 classes. The resolution of the image data varies and ranges from several hundred pixels. We crop the ImageNet data in all experiments to 224×224 pixels.

[5] https://www.uni-bremen.de/en/csl/research/motion-recognition.html

5.6.1.2 Hardware Simulation Setup

The proposed FantastIC4 was implemented in System Verilog, and corresponding behavioral and gate-level simulation was performed using the Mentor Graphics Simulator. Next, the FPGA version of the FantastIC4 was implemented using the Xilinx Vivado tool. Finally, we synthesized the place and routed the design to a Virtex Ultrascale FPGA on the XCVU440 device.

We synthesized the architecture using Synopsys Design Compiler (DC) under the ASIC version's GF 22nm FDSOI SLVT technology. Next, we placed and routed the design using Synopsys IC compiler (ICC2). After the sign-off and RC extraction using STARRC, we performed the timing closure using Synopsys Prime Time. Finally, we annotated the toggle rates from the gate-level simulation, dumped the toggling information into the Value Change Dump (VCD) file, and estimated the power using Prime Time.

The closest related training method to ours is EC2T, which trains sparse and ternary networks under an entropy-constrained regularizer [143]. However, our generalization allows us to train DNNs with more expressive power due to their ability to express 16 different cluster centers instead of only 3. Moreover, thanks to full-precision scaling factors, which can accommodate batch norm parameters, we expect our DNN models to be more robust to strong quantization plus the sparsification, consequently attaining better prediction for performance vs. compression trade-offs. Fig. 5.9 shows this phenomenon. Our DNN models trained with our entropy-constrained approach reach better "Pareto-Optimal" fronts regarding accuracy vs. sparsity than the EC2T method.

Furthermore, in Table 5.2, we show the prediction performance of our models on the datasets described above and summarize the results attained by other authors. We can see that we consistently attain similar or higher prediction performance than the previous work. Table 5.2 also shows the benefits of applying a hybrid compression scheme instead of the single compression format approach as proposed by previous work. Our compression scheme encodes each layer using the CSR, the simple Huffman code (or bitmask format), and the trivial 4-bit dense representation, and we choose the most compact representation between them. We see that we attain about 2.36 $\times$ boost in the compression gains on average as compared to the CSR-only approach proposed by [150, 152], and 1.77$\times$ higher compression rates than the trivial 4-bits dense format. These gains directly translate to reduced memory, off-to-on-chip data movement, and area requirements, stressing the importance of supporting multiple representations.

5.6.2 Benchmarking Hardware Efficiency

5.6.2.1 Results on MLPs

We benchmarked FantastIC4 hardware performance on the fully connected layers of several popular models such as EfficientNet-B0, MobileNet-v3 and ResNet-50. Furthermore, we benchmarked the end-to-end inference efficiency of two of our custom and fully hardware-conform MLPs, trained for the Google Speech Commands and Hand Gesture Recognition task. Both MLP models, which we named *MLP-GSC* & *MLP-HR* respectively, reach SoA prediction performance on their tasks.

Table 5.2: Comparison of the FantastIC4-quantization approach vs previous SoA 4-bit quantization techniques. We report two results for each network, one showing the highest accuracies we attained and the other with the highest compression ratios. Our models, as well as the best results, are highlighted in bold. All models belonging to the same row block have the same architecture, except the Google Speech Command and Hand Gesture Recognition datasets. Unless otherwise specified, all approaches quantize all network layers, including input- and output layers, excluding batch normalization- and bias parameters [32] ©2021 IEEE.

Model	Org. Acc. (%)	Acc.	Size (MB)	CR[c]	CSR[d]
ImageNet					
EfficientNet-B0	76.43	**75.01**	21.15	7.62	3.31
EfficientNet-B0	76.43	74.08	21.15	**8.25**	3.91
LSQ+ [135]	76.10	73.80	21.15	7.48	2.59
ResNet-50	76.15	**75.66**	102.23	8.21	3.50
ResNet-50	76.15	75.29	102.23	**9.97**	4.50
PWLQ [136]†	76.13	75.62	102.23	7.86	2.64
KURE [137]†	76.30	75.60	102.23	7.88‡	2.64‡
ResNet-34$_{IO}$†	73.30	72.98	87.19	7.80	4.32
ResNet-34$_{I}$†	73.30	72.86	87.19	**9.30**	4.37
QIL [138]†	73.70	**73.70**	87.19	6.82	2.65
DSQ [139]†	73.80	72.76	87.19	6.82	2.65
CIFAR-10					
ResNet-20	91.67	91.60	1.08	8.43	3.92
ResNet-20	91.67	91.15	1.08	**16.23**	11.31
SLB [140]†	92.10	**92.10**	1.08	7.64	2.62
GWS [141]†	92.20	91.46	1.08	7.72‡	2.62‡
MNIST					
LeNet-300-100	98.70	**98.63**	1.07	13.31	7.62
LeNet-300-100	98.70	98.16	1.07	**29.31**	19.81
Google Speech Commands					
MLP-GSC	91.00	**91.19**	2.57	10.88	5.55
MLP-GSC	91.00	90.41	2.57	**13.59**	7.99
HE [**HE**]	86.40	86.40	0.2	-	-
Hand Gesture Recognition					
MLP-HR	88.50	**88.33**	1.30	8.51	3.96
MLP-HR	88.50	87.22	1.30	**13.57**	8.35
HMM [20]	74.30	74.30	-	-	-

[a] Compression ratio is defined as the ratio of the full precision model size to the quantized model size, where FantastIC4 stores each layer in its optimal format, which is CSR, bitmask format, or the trivial 4-bit dense format.

[b] Compression ratio defined as the ratio of the full-precision to the quantized model size, where each layer is stored in CSR format.

† QIL and DSQ use full-precision (32-bit) for the first and last layer, PWLQ and SLB use full-precision for the first layer and KURE and GWS provide no information about first/last layer quantization. Our ResNet-34$_{IO}$ benchmark has 32bit input- and output layers, and the ResNet-34$_{I}$ benchmark has a 32bit input layer.

Table 5.3 shows the resource utilization breakdown of our FantastIC4 accelerator for different DNN models. The proclaimed results are based on the post-implementation results

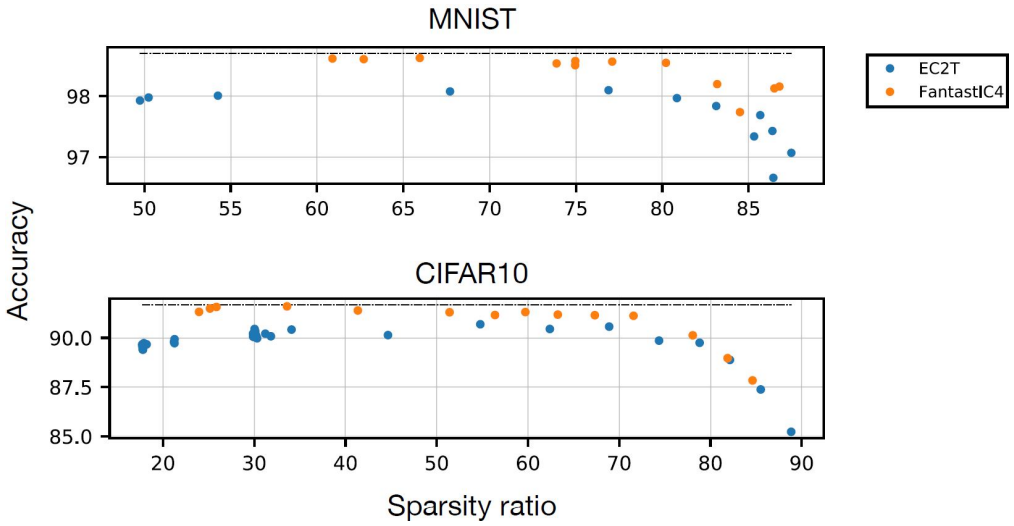


Figure 5.9: Accuracy as a function of the sparsity ratio of different DNN models. (top) LeNet-300-100 model trained on the MNIST dataset by the previous method EC2T [143], compared to FantastIC4 generalized form of entropy-constrained training method. (bottom) same as top, but for ResNet20 trained on the CIFAR10 dataset [32] ©2021 IEEE.

Table 5.3: FantastIC4 resource utilization breakdown for different DNN Models on a Virtex Ultrascale FPGA. Here, BR stands for BRAMs, FF for flip flops, LUT for look-up tables, DSP for digital signal processing, and LR stands for LUTRAMs [32] ©2021 IEEE.

Modules	MLP-HR					EfficientNet-B0					MobileNet-V3					ResNet-50					MLP-GSC				
	LUT	FF	BR	DSP	LR	LUT	FF	BR	DSP	LR	LUT	FF	BR	DSP	LR	LUT	FF	BR	DSP	LR	LUT	FF	BR	DSP	LR
CSR to BM	5K	255	0	0	0	5K	255	0	0	0	5K	255	0	0	0	5K	255	0	0	0	5K	255	0	0	0
Wt ID Gen	0	512	0	0	0	0	512	0	0	0	0	512	0	0	0	0	512	0	0	0	0	512	0	0	0
MAC Array	565	153	0	4	0	570	158	0	4	0	568	156	0	4	0	580	162	0	4	0	566	153	0	4	0
Fixed point to Float point Op	707	483	8	4	0	717	491	8	4	0	711	17	12	4	0	719	500	16	4	0	709	484	8	4	0
Adder tree	35K	4K	0	0	0	35K	4K	0	0	0	35K	4K	0	0	0	35K	4K	0	0	0	35K	4K	0	0	0
FIFO Module	53K	6K	0	0	8K	103K	119K	0	0	160K	830K	95K	0	0	128K	1661K	190K	0	0	256K	53K	6K	0	0	8K
BM Memory	21	821	8	0	0	38	842	36	0	0	33	834	31	0	0	48	821	63	0	0	28	822	8	0	0
Total	95K	12K	16	8	8K	108K	125K	44	8	160K	872K	101K	43	8	128K	1703K	196K	79	8	256K	95K	12K	16	8	8K

Table 5.4: FantastIC4 final resource utilization [32] ©2021 IEEE.

Resource	LUT	LUTRAM	FF	BRAMs	DSP
Used	1703,187	128,000	196,909	79	8
Available	2532,960	459,360	5065,920	2,520	2,880
Utilization	67.24%	27.86%	3.88%	3.13%	0.27%

from Xilinx Vivado 2018.2. The activations values are quantized down to 8-bit precision, whereas the four basis weights use a precision of 16-bits. This configuration was found to be accurate enough to perform the inference without harming the prediction performance of the models. As shown in Table 5.3, we consume the lowest resources among all the accelerators reported so far that perform on FC layers. Here, we engage fixed and floating-point operations for faster processing and improved accuracy during the inference. As a result, we consume just 8 DSPs to perform the computation, significantly reducing dynamic power consumption. Moreover, very few BRAMs are used in the inference operation due to the extreme quantization and compression. Instead, the LUTRAMs are explicitly used to store the weightIDs inside the FIFOs. As one can see, the floating-point operations utilize the lowest resources on the FPGA

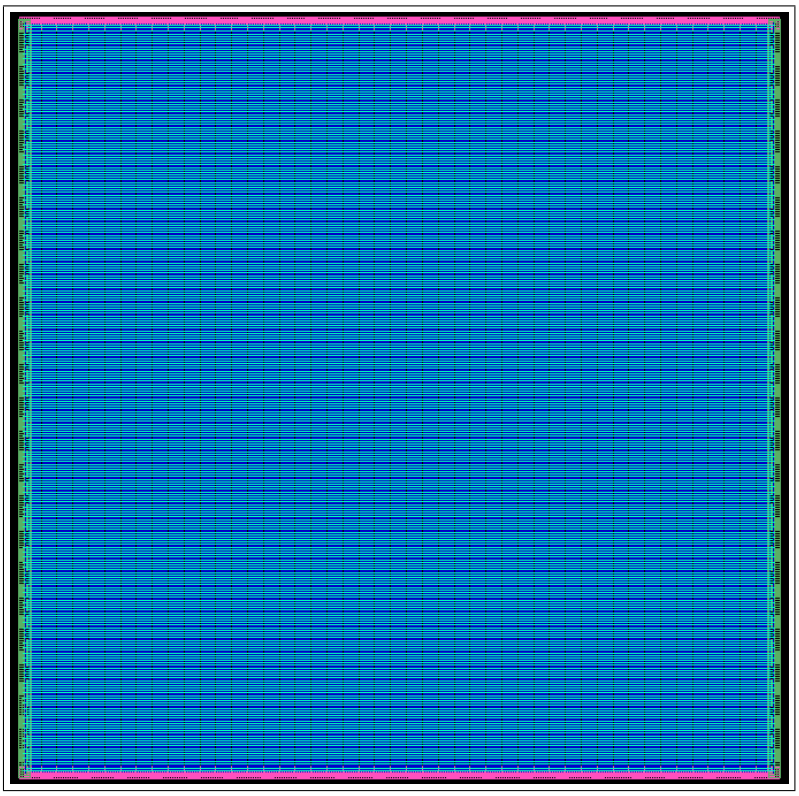


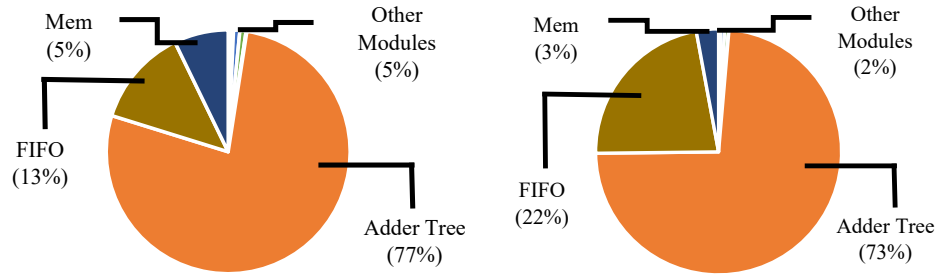
Figure 5.10: Layout view [32] ©2021 IEEE.

Table 5.5: Post layout results of the ASIC version [32] ©2021 IEEE.

Technology	GF 22nm FDSOI SLVT
Chip Size	1mm × 1.2mm
Core Area	$800\mu m \times 800\mu m$
Core Voltage	0.88V
Memory Type	SRAM (10KB)
Total Gate Count	961K
Frequency	800MHz
Precision	Fixed 16bit
Power	454mW
Latency	1.31μs
Performance	9.158 TOPS
Performance/W	20.17 TOPS/W
Energy	595 nJ
DNN Models Inferenced	MLP-HR and MLP-GSC

chip due to the enhanced data flow modeling. Table 5.4 shows the final resource utilization summary.

We further evaluated the ASIC version of FantastIC4 on a 22nm process node with a clock frequency of 800MHz. Table 5.5 reports the layout version, and its layout view is shown in Fig. 5.10, the total area of our processor was found to be 1mm × 1.2mm.



(a) Area Breakdown (b) Power Breakdown

Figure 5.11: Area and power breakdown of FantastIC4 ASIC version [32] ©2021 IEEE.

5.6.2.2 Power Consumption, Latency and Throughput on the FPGA

The FantastIC4 accelerator is highly energy-efficient due to the low-weight storage and the static activations inside the adder tree. The static activations inside the adder tree reduce the total power consumption by 15×, as the decreasing data movement consumed around 64mW of dynamic power compared to the conventional SRAM access, which had 960mW of power consumption. We measured the power consumption for different DNN models. Throughout the inference, the static power consumption was significantly higher than the dynamic power consumption, as the static power consumed on the XCVU440 FPGA was 2.856W. The total power measured from the inference of MLP-HR was 3.472W, EfficientNet-B0 was 10.14W, ResNet-50 was 12.34W, MobileNet-V3 was 8.46W, and MLP-GSC was 3.6W. The average latency measurement of each layer for MLP-HR was 6.45μs, EfficientNet-B0 was 8.6μs, MobileNet-V3 was 6.3μs, ResNet-50 was 10.2μs, and MLP-GSC was 7.2μs. To infer our entire custom DNN model, we had a latency of 72μs for MLP-HR and 80μs for MLP-GSC. The overall throughput measurement was 2.45TOPS, as the processing unit remains constant irrespective of the DNN model under inference. On average, the total off-chip to on-chip data movement was saved by 10.55× compared to the parameters' original (non-compressed) representation. Furthermore, due to our on-chip support of hybrid compressed representations, we could boost the savings by 2× compared to the compressed formats proposed by [150] and [152].

Similarly, for the ASIC version, as shown in Table 5.5, we could achieve a peak performance throughput of 13.1 TOPS and a performance/watt of 28.87 TOPS/W. The performance/watt is the unit used to evaluate the total number of operations performed by the inference architecture per power consumption unit. The total latency to perform the inference was 1.31μs for the MLP-HR model and 1.37μs for the MLP-GSC model. Since we first perform all the accumulation on the adder tree and then perform the MAC operation, we significantly save the resources and power required for computations by 2.7×. For example, an array of 256 MAC units with 16-bit width consumes an area of 346.58μm × 346.58μm, whereas the same ACM unit will consume an area of 216.54μm × 216.54μm. Similarly, an array of 256 MAC units consumes a power of 101.23 mW, and a variety of 256 ACM units consumes a power of 40.46 mW. So, by our ACM technique, we save a total area of around 39% and power of around 40%. The ASIC version's location and power breakdown are shown in Fig. 5.11. In

Table 5.6: Performance comparison with other SoA FPGA accelerators [32] ©2021 IEEE.

Parameters	Dinelli [170]	Ours
Device	XCVU65	XCVU440
Benchmark	GSC	GSC
Quantization	Fixed-16	Fixed-16
Sparsity	N/A	60%
Accuracy	90.23%	91%
Throughput (TOPS)	N/A	2.45
Throughput/W (GOPS/W)	N/A	198.54
Static Power (W)	0.626	2.856
Dynamic Power (W)	1.235	0.744
Total Power (W)	1.861	3.6
Latency (μs)	570	80
Energy (mJ)	1.06	0.288
Frequency (MHz)	78.4	150

Table 5.7: Performance comparison with other SoA ASIC compression-based accelerators [32] ©2021 IEEE.

Platform	EIE [150]	Eyeriss V2 [152]	Thinker [171]	Our's
Technology (nm)	65	65	65	22
Frequency (MHz)	800	200	200	800
Precision	Fixed-16	Fixed-16	Fixed-8/16	Fixed-16
Throughput (GOPS)	572	858.62	368.4	9158.65
Power (mW)	590	606	290	454
Power Efficiency (GOPS/W)	969.49	1416.87	1270.34	20173.23
Area (mm^2)	40.8	N/A	19.6	1.2
Area Efficiency ($GOPS/mm^2$)	14.02	N/A	18.79	7632.208

most areas, the adder tree and the FIFOs dominate power consumption as they form the core part of the architecture.

5.6.3 Comparison to previous work

Here, we compare the performance of other SoA accelerators on FPGA that work on multi-layer perceptrons with a benchmark on the GSC dataset. The Keyword Spotting (KWS) accelerator [170] was the closest FPGA accelerator that benchmarked on GSC, so for a fair comparison, we are comparing it with this accelerator. The KWS accelerator [170] also quantized their DNN models, implemented the entire architecture using on-chip memories, and benchmarked the results on different Xilinx and Intel FPGA devices. Table 5.6 shows the comparison results. Here, we are mainly benchmarking for sparsity, accuracy, throughput, and power consumption. We evaluated the performance of our accelerator on our custom MLP-GSC, as our custom-built network had more sparsity and higher accuracy for the KWS application. Our FantastIC4 accelerator has an overall throughput of 2.45 TOPS due to the parallel execution of the adder tree and the MAC array and lower clock cycle requirement for the floating-point operations. We have 50× lower dynamic power consumption compared to [170] due to the static activations inside the adder tree, lesser number of multiplications, and pipelined approach with the floating point operations. Regarding latency, we are 14× faster to

Table 5.8: Performance comparison with other SoA ASIC KWS Accelerators [32] ©2021 IEEE.

Platform	EERA-ASR [172]	Guo [173]	Our's
Technology (nm)	28	65	22
Frequency (MHz)	400	75	800
Latency (us)	N/A	127.3	1.31
Keywords Number	20	10	10
Dataset	GSC		
Accuracy	91.88%	90.20%	91%
Throughput (GOPS)	179.2	614.4	9158.65
Power (mW)	54	52.5	454
Energy(nJ)	N/A	6683.25	594.74
Power Efficiency (TOPS/W)	3.31	11.7	20.17
Area(mm^2)	3.34	6.2	1.2
Area Efficiency ($GOPS/mm^2$)	53.65	52.51	7632.208

infer one complete network that works on the KWS application. Regarding energy efficiency, we are 27.16× better compared to the other accelerator.

For the ASIC version, arguably the closest related work to FantastIC4 are EIE [150] & EyerissV2 [152] since both accelerators also leverage on compressed representations of the DNNs parameters. We stress that more recent accelerators exploiting compressed representations exist, such as [151]. However, these were optimized for convolutional layers, whereas FantastIC4 optimizes the execution of fully-connected layers.

In the following, we provide benchmarks across different accelerators for each component, as shown in Table 5.7. In the throughput comparison, FantastIC4 is better than EIE by 16×, Eyeriss v2 by 15×, and Thinker by 31×. Regarding power efficiency, FantastIC4 outperforms EIE by 20×, Eyeriss v2 by 14×, and Thinker by 16×. We could not compare our accelerator with Eyeriss v2 because the Eyeriss v2 area is reported in terms of the total number of gates. So, by comparing the total gates, we are smaller by 2.9×. However, for other accelerators, we are better than EIE by 544× and Thinker by 406×.

Table 5.8, we compare our accelerators with the other state of art ASIC KWS accelerators. We are comparing FantastIC4 with EERA-ASR[172] and RNN-based speech recognition processor [173]. Both the processors work with the same Google Speech Command dataset. Compared to other works, we have a better throughput by 51× and 14×. Similarly, we are more power efficient by 6× and 1.8×, respectively. Regarding area efficiency, we are efficient by 142× concerning [172] and 145× concerning [173].

Finally, a few of the things that FantastIC4 can improve upon is by making the architecture compatible for the inference of the convolution layers of the DNN and further improving the performance/watt by reducing the dynamic power consumption in both the FPGA and ASIC inference.

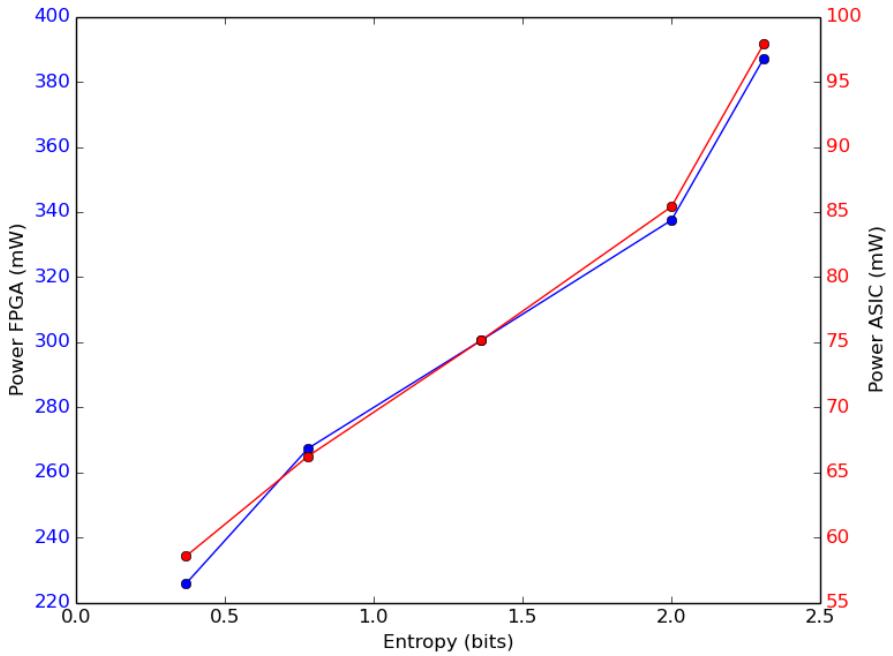


Figure 5.12: Power consumption of our MLP-HR model as a function of its entropy distribution. (blue) Dynamic power consumption measured on an FPGA, (red) measured on ASIC simulation [32] ©2021 IEEE.

5.6.4 Ablation Study: Execution efficiency of the models as a function of their entropy

In Section III, we argued that one of our significant contributions is that FantastIC4 hardware architecture is specially designed to exploit low-entropy statistics of the weight parameters. Thus, we should expect the execution efficiency of DNN models to increase as the entropy of the weight parameters decreases. Figure 5.12 shows precisely this trend. In this experiment, we measure the power efficiency of our MLP-HR model at different overall entropy levels of the model. To perform this study, we ran our post-layout simulation (ASIC) and post-implementation timing simulation (FPGA) to generate the corresponding Value Change Dump (VCD) for ASIC and Switching Activity Interchange Format (SAIF) for FPGA. Using these files, we measured the dynamic (vector-based) power consumption on Synopsys PrimeTime and Vivado. Based on the measurement, the power consumption decreases quasi-linearly with the entropy of the model. Again, this trend is because FantastIC4 supports (1) the efficient processing of compressed representations of the weight parameters, (2) the efficient computation of 4-bit non-zero values, and (3) the efficient loading of repeated values from the FIFOs, all being properties that become more and more predominant as the entropy of the model's parameters decreases.

5.7 Summary

This chapter proposed a software-hardware optimization paradigm for maximally increasing the area and power efficiency of MLP models with SoA predictive performance. First, we introduce a novel entropy-constrained training method for making the models compressible in size, which,

in combination with FantastIC4s support for the efficient on-chip execution of multiple compact representations, can boost the data movement efficiency of the parameters by up to 29× (on average 10.55× across different models) as compared to the original models, and by 2× as compared to the previous compression approaches. In addition, our training algorithm renders the models robust to 4-bit quantization while inducing sparsity, properties that FantastIC4 exploits to further increase the power efficiency by 2.7× and area efficiency by 2.6×. Finally, it implements an activation stationary data movement paradigm, increasing the on-chip data movement efficiency of the activation values by 15×. FantastIC4 was implemented on a Virtual Ultrascale FPGA XCVU440 device. The experimental results show an overall throughput of 2.45 TOPS with a total power consumption of 3.6W. Furthermore, we achieved the lowest resource utilization for a MLPs inference by consuming 67.24% of LUTs, 27.86% of LUTRAMs, 3.88% of FFs, 3.13% of BRAMs, and 0.27% of DSPs. This is the first FPGA accelerator to achieve high throughput using a few resources and power. We further benchmarked our FantastIC4 on a 22nm process. The ASIC version achieved a total power efficiency of 20.17 TOPS/W and latency of 1.31μs per layer inference of the GSC dataset. FantastIC4 outperforms other SoA GSC accelerators by 51× in terms of throughput and 145× in area efficiency.

6

Conclusion

6.1 Summary of the Contributions

Research on DNN architectures is becoming immensely popular with its wide performance and applicability. Despite the high amount of related works in this ever-booming field, this thesis makes a few solid contributions to the DNN hardware research community:

- Energy Efficient Dataflows: As we have seen how data movement inside the DNN processor is a very crucial attribute, **this thesis systematically exploits the multi-level storage in the form of SRAM and scratch pad usage as explained in Chapter 4 with an optimized pipelined dataflow to achieve energy efficiency**. Furthermore, unlike previous works, which perform one-size-fits-all dataflows regardless of the DNN shapes and sizes, we have shown quantitatively that better results can be achieved when the dataflow is dynamically adapted to the DNN data structures.

- Efficient PE utilization and ACM Engine for high-performance: The biggest bottleneck in the DNNs processing is the data reuse variation, which can lead to inefficient throughput due to low PE utilization. **This thesis describes how to reuse the data and improve the performance throughput by avoiding the traditional MAC engine and replacing it with the ACM engine, as explained in Chapter 5. ACM engines avoid underutilization of PEs. Also, it supports the highly sparse DNN networks by turning off the computation to save the dynamic power consumption when zero weights or activations are loaded into the PE unit.** Unlike the earlier works that mainly concentrate on the lower performance bounds, such as the peak performance or the number of active PEs to evaluate the hardware performance. We have quantitatively shown the effect of the ACM when compared to the traditional MAC units and the need for approximate multiplication to save power consumption.

- Energy efficient dataflows: The RS dataflow in the multi-bit accelerator in Chapter 4 and the pipelined dataflow of FantastIC4 in Chapter 5 demonstrated the need for highly adaptive dataflows that optimizes for the energy efficiency of the DNN models.

- Dynamic truncation of the PSums: The multi-bit accelerator sets an example of how dynamically the PSums data can be truncated that serves the wide variety of DNNs in one design by inferencing on the multiple bits. **The developed accelerator proves that there is no need to have a fixed-bit architecture to achieve high accuracy, and it quantitatively proves that with reduced bit-width, we can achieve low power, high throughput, low resource utilization, and high accuracy.**

- Hardware-software co-design processor: FantastIC4 processor showed that hardware-software co-designing could achieve both computational efficiency and an increased model capacity. It also quantitatively proved that compression could increase the throughput and fit larger DNN models into the on-chip SRAM. In addition, this processor design also proved that the fixed and floating-point operations could be supported on the same architecture.

Overall, this thesis demonstrates the need for the hardware-software co-design to achieve an efficient performance, power, and area processor. Achieving the proper balance between the two methods will open more opportunities in the AI hardware field to be applied in real-world applications with energy and performance constraints.

6.2 Drawbacks

Although the thesis adds significantly to other SoA architectures, Chapter 4 and Chapter 5 still can improve on some aspects of their architectures. For example, the multi-bit architecture supports only the point-wise convolutions, and it can be extended to support the depth-wise convolution and the swish activation function. In addition, the latency for inference can be further improved for the next generation DNN models.

Similarly, the FantastIC4 architecture can be extended further to the convolution layers as it supports only the FC layers.

6.3 Future Work

As research on DNNs continues, more challenges and opportunities will arise in developing DNN accelerators.

- *Hardware Friendly DNN Designs:* The design of DNN hardware architectures is lagging compared to DNN model development. Since algorithm designers and hardware architects depend on each other for efficient design development, this creates a significant knowledge gap. Research and development on providing instant feedback on the processing throughput and energy efficiency in the design or during the training will be beneficial to bridge the gap and yield hardware-friendly DNNs.

- *Hardware Architecture for Training:* This thesis aims to investigate the inference part of DNN models. Using hardware to train the models would reduce overall computation time and benefit applications that require privacy and customization. Training is mainly done offline due to the increased hardware resources and run time. Building efficient hardware for training will help open up more opportunities in the AI field.
- *Generalization to other fields:* Most of the design principles presented in this thesis apply to different fields. For example, the computations to handle the sparsity in DNNs can benefit sparse linear algebra. Furthermore, GPUs can cue from the DNN data flow as the computation flow remains unchanged.

References

[1] Murray Campbell, A. Joseph Hoane, and Feng Hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1-2 (2002), pp. 57–83. ISSN: 00043702. DOI: 10.1016/S0004-3702(01)00129-1.

[2] David Ferrucci et al. "Building watson: An overview of the deepQA project". In: *AI Magazine* 31.3 (2011), pp. 59–79. ISSN: 07384602. DOI: 10.1609/aimag.v31i3.2303.

[3] "Siri does more than ever.Even before you ask ". In: (2009). URL: https://www.apple.com/in/siri/.

[4] "Why this matters ". In: (2009). URL: https://waymo.com/.

[5] "Microsoft Corotana ". In: (2009). URL: https://www.microsoft.com/en-us/cortana.

[6] Iqbal H Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2 (2021). DOI: 10.1007/s42979-021-00592-x.

[7] R. Konieczny and R. Idczak. "Mössbauer study of Fe-Re alloys prepared by mechanical alloying". In: *Hyperfine Interactions* 237.1 (2016), pp. 1–8. ISSN: 15729540. DOI: 10.1007/s10751-016-1232-6.

[8] Geoffrey. Hinton and Terrence. Sejnowski. "Unsupervised learning: Foundations of neural computation". In: *Computers & Mathematics with Applications* 38.5-6 (1999), p. 256. ISSN: 08981221. DOI: 10.1016/s0898-1221(99)90165-7.

[9] Xiaojin Zhu. "Semi-Supervised Learning Literature Survey Contents". In: *SciencesNew York* 10.1530 (2008), p. 10. ISSN: 0002-7863. DOI: 10.1.1.146.2352. arXiv: 1412.6596. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.99.9681%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[10] Leslie. Kaelbling, Michael. Littman, and Andrew. Moore. "Reinforcement Learning: A Survey". In: *Journal of Artificail Intelligence Research* (1996). ISSN: 16747321. DOI: 10.1007/s11431-012-4938-y.

[11] S. Bozinovski. "A self-learning system using secondary reinforcement". In: *Proceedings of the Sixth European Meeting on Cybernetics and Systems Research* (1982), pp. 397–402.

[12] A.M. Tillmann. "On the Computational Intractability of Exact and Approximate Dictionary Learning". In: *IEEE Signal Processing Letters* (2015), pp. 45–49.

[13] A. Zimek and E. Schubert. "Outlier Detection". In: *Encyclopedia of Database Systems, Springer New York* (2017), pp. 1–5. DOI: 10.1007/978-1-4899-7993-3_80719-1.

[14] A. Moringen et al. “Attention-Based Robot Learning of Haptic Interaction”. In: *International Conference on Human Haptic Sensing and Touch Enabled Computer Applications* 12270 (2020), pp. 462–470. DOI: 10.1007/978-3-030-58147-3_51.

[15] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 14764687. DOI: 10.1038/nature14539.

[16] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778. ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances In Neural Information Processing Systems* (2012), pp. 1–9. ISSN: 10495258. DOI: http://dx.doi.org/10.1016/j.protcy.2014.09.007. arXiv: 1102.0183.

[18] Li Deng et al. “Recent Advances in Deep Learning for Speech Reasearch at Microsoft Li”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 8604–8608.

[19] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *arXiv:1804.03209 [cs]* (Apr. 2018). arXiv: 1804.03209.

[20] Marcus Georgi, Christoph Amma, and Tanja Schultz. “Recognizing Hand and Finger Gestures with IMU based Motion and EMG based Muscle Activity Sensing”. In: *International Conference on Bio-inspired Systems and Signal Processing* (2015), pp. 99–108. DOI: 10.5220/0005276900990108.

[21] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961. URL: http://dx.doi.org/10.1038/nature16961.

[22] Kevin Kiningham, Michael Graczyk, and Athul Ramkumar. “Design and Analysis of a Hardware CNN Accelerator”. In: (2016), p. 33. URL: http://cs231n.stanford.edu/reports/2017/pdfs/116.pdf.

[23] Roger Fingas. “Tim Cook says AI augmented reality are core technologies in Apple’s future”. In: (2016). URL: https://appleinsider.com/articles/16/08/14/tim-cook-says-ai-augmented-reality-are-core-technologies-in-apples-future.

[24] “Why Truly Smart Future Tech Needs Radically New AI Chips ”. In: (2020). URL: https://insidebigdata.com/2020/03/17/why-truly-smart-future-tech-needs-radically-new-ai-chips.

[25] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 07-12-June (2015), pp. 1–9. ISSN: 10636919. DOI: 10.1109/CVPR.2015.7298594. arXiv: arXiv:1409.4842v1.

[26] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: (2014), pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[27] S. Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), pp. 243–254.

[28] Y.H Chen, J Emer, and V Sze. “Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators”. In: *IEEE Micro* (2015). DOI: 10.1109/MM.2017.54.

[29] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. ISSN: 00189219. DOI: 10.1109/JPROC.2017.2761740. arXiv: 1703.09039.

[30] Suhas Shivapakash et al. “A Power Efficient Multi-Bit Accelerator for Memory Prohibitive Deep Neural Networks”. In: *IEEE Circuits and System Conference (ISCAS)* (2020).

[31] Suhas Shivapakash et al. “A Power Efficiency Enhancements of a Multi-Bit Accelerator for Memory Prohibitive Deep Neural Networks”. In: *IEEE Open Journal of Circuits and Systems* (2020), pp. 161–170. DOI: 10.1109/OJCAS.2020.3047225.

[32] Simon Wiedemann et al. “FantastIC4: A Hardware-Software Co-Design Approach for Efficiently Running 4bit-Compact Multilayer Perceptrons”. In: (2020), pp. 1–15. arXiv: 2012.11331. URL: http://arxiv.org/abs/2012.11331.

[33] Dan Ciregan, Ueli Meier, and Jurgen Schmidhuber. “Multi-column deep neural networks for image classification”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2012), pp. 3642–3649. ISSN: 10636919. DOI: 10.1109/CVPR.2012.6248110. arXiv: 1202.2745.

[34] Robert Geirhos et al. “Comparing deep neural networks against humans: object recognition when the signal gets weaker”. In: (2017). ISSN: 2331-8422. arXiv: 1706.06969. URL: http://arxiv.org/abs/1706.06969.

[35] Niall O’Mahony et al. “Deep Learning vs. Traditional Computer Vision”. In: *Advances in Intelligent Systems and Computing* 943.Cv (2020), pp. 128–144. ISSN: 21945365. DOI: 10.1007/978-3-030-17795-9_10. arXiv: 1910.13796.

[36] J Han and C Moraga. “The influence of the sigmoid function parameters on the speed of backpropagation learning”. In: *International Workshop on Artificial Neural Networks* (2005).

[37] M Chandra. “Hardware Implementation of Hyperbolic Tangent Function using Catmull-Rom Spline Interpolation”. In: *CoRR* abs/2007.13516 (2020).

[38] Abien Fred Agarap. “Deep Learning using Rectified Linear Units (ReLU)”. In: 1 (2018), pp. 2–8. arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[39] Guanghua Ren et al. “A modified Elman neural network with a new learning rate scheme”. In: *Neurocomputing* 286.April (2018), pp. 11–18. ISSN: 18728286. DOI: 10.1016/j.neucom.2018.01.046.

[40] A Wysocki and M Ławryńczuk. “Jordan neural network for modelling and predictive control of dynamic systems”. In: *20th International Conference on Methods and Models in Automation and Robotics (MMAR)* (2015). DOI: 10.1109/MMAR.2015.7283862.

[41] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biophysics* (Dec. 1943), pp. 115–133. DOI: https://doi.org/10.1007/BF02478259.

[42] Y LeCun et al. “Handwritten Digit Recognition: Applications”. In: *IEEE Communications Magazine* 27.November (1989), pp. 41–46.

[43] J Woodhouse. “Big, big, big data: higher and higher resolution video surveillance”. In: (2016). URL: https://www.edge-ai-vision.com/2016/01/big-big-big-data-higher-and-higher-resolution-video-surveillance/.

[44] K Simonyan and A Zisserman. “Two-stream convolutional networks for action recognition in videos”. In: *NIPS* (2014).

[45] J Long et al. “Fully Convolutional Networks for Semantic Segmentation”. In: *CVPR* (2015).

[46] Dong Wang and Thomas Fang Zheng. “Transfer Learning for Speech and Language Processing”. In: *arXiv* (2015). arXiv: arXiv:1511.06066v1.

[47] Henry Zhu et al. “The Ingredients of Real-World Robotic Reinforcement Learning”. In: (2020), pp. 1–20. arXiv: arXiv:2004.12570v1.

[48] David Silver et al. “Mastering the Game of Go without Human Knowledge”. In: *Nature* (2017). DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[49] Justin Ker et al. “Deep Learning Applications in Medical Image Analysis”. In: *IEEE Access* December (2017). DOI: 10.1109/ACCESS.2017.2788044.

[50] Huaizheng Zhang et al. “Hysia: Serving DNN-Based Video-to-Retail Applications in Cloud”. In: (2020), pp. 4457–4460. URL: https://doi.org/10.1145/3394171.3414536.

[51] S. Wiedemann et al. “DeepCABAC: A universal compression algorithm for deep neural networks”. In: *IEEE Journal of Selected Topics in Signal Processing* 14.4 (2020), pp. 700–714.

[52] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”. In: *Proceedings of the IEEE International Conference on Computer Vision* (2016), pp. 1–13. arXiv: 1602.07360. URL: http://arxiv.org/abs/1602.07360.

[53] Hongyu Zhu et al. “Benchmarking and Analyzing Deep Neural Network Training”. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)* (2018), pp. 88–100. DOI: 10.1109/IISWC.2018.8573476.

[54] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (2016).

[55] Alex Sherstinsky. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network”. In: *CoRR* abs/1808.03314 (2018). arXiv: 1808.03314. URL: http://arxiv.org/abs/1808.03314.

[56] Andre Xian Ming Chang and Eugenio Culurciello. "Hardware accelerators for recurrent neural networks on FPGA". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)* (2017), pp. 1–4. DOI: 10.1109/ISCAS.2017.8050816.

[57] Kevin Siu et al. "Memory Requirements for Convolutional Neural Network Hardware Accelerators". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)* (2018), pp. 111–121. DOI: 10.1109/IISWC.2018.8573527.

[58] Youngeun Kwon and Minsoo Rhu. "Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning". In: *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-51 (2018), pp. 148–161. DOI: 10.1109/MICRO.2018.00021. URL: https://doi.org/10.1109/MICRO.2018.00021.

[59] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression". In: (2017). arXiv: 1707.06342.

[60] Sourav Bhattacharya and Nicholas D. Lane. "Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables". In: *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. SenSys '16 (2016), pp. 176–189. DOI: 10.1145/2994551.2994564. URL: https://doi.org/10.1145/2994551.2994564.

[61] Neena Aloysius and M. Geetha. "A review on deep convolutional neural networks". In: *2017 International Conference on Communication and Signal Processing (ICCSP)* (2017), pp. 0588–0592. DOI: 10.1109/ICCSP.2017.8286426.

[62] Lukas Cavigelli and Luca Benini. "Origami: A 803 GOp/s/W Convolutional Network Accelerator". In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.11 (2015), pp. 2461–2475. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2016.2592330. arXiv: 1512.04295.

[63] Dubravko Majetic et al. "Neural network without bias neuron for hidden layer". In: *Annals of DAAAM and Proceedings of the International DAAAM Symposium* (Jan. 2005), pp. 239–240.

[64] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[65] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22nd ACM International Conference on Multimedia*. MM '14 (2014), pp. 675–678. DOI: 10.1145/2647868.2654889. URL: https://doi.org/10.1145/2647868.2654889.

[66] Vinod Nair and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10 (2010), pp. 807–814. URL: https://dl.acm.org/doi/10.5555/3104322.3104425.

[67] A.L Maas, A.Y Hannun, and A.Y Ng. "Rectifier non linearities improve neural network acoustic models". In: *ICML* (2013).

[68] K He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *ICCV* (2015).

[69] D.A Clevert, T Unterthiner, and S Hochreiter. “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)”. In: *ICLR* (2016).

[70] X Zhang et al. “Improving deep neural network acoustic models using generalized maxout networks”. In: *ICASSP* (2014).

[71] Y Zhang et al. “Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks”. In: *Interspeech* (2016).

[72] P Sermanet et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *ICLR* (2016).

[73] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[74] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2018), pp. 4510–4520. ISSN: 10636919. DOI: 10.1109/CVPR.2018.00474. arXiv: 1801.04381.

[75] Andrew Howard et al. “Searching for mobileNetV3”. In: *Proceedings of the IEEE International Conference on Computer Vision* 2019-Octob (2019), pp. 1314–1324. ISSN: 15505499. DOI: 10.1109/ICCV.2019.00140. arXiv: 1905.02244.

[76] Mingxing Tan and Quoc V. Le. “EfficientNet: Rethinking model scaling for convolutional neural networks”. In: *36th International Conference on Machine Learning, ICML 2019* 2019-June (2019), pp. 10691–10700. arXiv: 1905.11946.

[77] L Wan et al. “Regularization of neural networks using dropconnect”. In: *ICM* (2013).

[78] Li Deng. “The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]”. In: *Signal Processing Magazine, IEEE* 29 (Nov. 2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[79] Rasim Caner Çalik and M. Fatih Demirci. “Cifar-10 Image Classification with Convolutional Neural Networks for Embedded Systems”. In: *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)* (2018), pp. 1–2. DOI: 10.1109/AICCSA.2018.8612873.

[80] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: (2015). arXiv: 1409.0575.

[81] S Condon. “Facebook unveils Big Basin, new server geared for deep learning”. In: *ZDNet* (2017).

[82] M Mathieu, M Henaff, and Y LeCun. “Fast training of convolutional networks through FFTs”. In: *ICLR* (2014).

[83] M Mathieu and F Fleuret. “Exact acceleration of linear object detectors”. In: *ECCV* (2012).

[84] Aleksandar Zlateski et al. “FFT Convolutions are Faster than Winograd on Modern CPUs, Here is Why”. In: (2018). arXiv: 1809.07851.

[85] Sharan Chetlur et al. “cuDNN: Efficient Primitives for Deep Learning”. In: (2014). arXiv: 1410.0759.

[86] Clément Farabet et al. “NeuFlow: A runtime reconfigurable dataflow processor for vision”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (2011). ISSN: 21607508. DOI: 10.1109/CVPRW.2011.5981829.

[87] Chen Zhang et al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays(FPGA)* (2015), pp. 161–170. arXiv: 2684746.2689060 [10.1145].

[88] Yunji Chen et al. “DaDianNao: A Machine-Learning Supercomputer”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[89] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. “Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators”. In: *IEEE Micro* 37.3 (2017), pp. 12–21. DOI: 10.1109/MM.2017.54.

[90] Yu Hsin Chen et al. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. ISSN: 00189200. DOI: 10.1109/JSSC.2016.2616357.

[91] Rambus Press. “Reduced-Precision Computation for Neural Network Training”. In: (2020). URL: https://www.rambus.com/blogs/reduced-precision-computation-for-neural-network-training.

[92] Joe Jeddeloh and Brent Keeth. “Hybrid memory cube new DRAM architecture increases density and performance”. In: *Symposium on VLSI Technology (VLSIT)* (2012), pp. 87–88. DOI: 10.1109/VLSIT.2012.6242474.

[93] Sangyun Oh et al. “Automated Log-Scale Quantization for Low-Cost Deep Neural Networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), pp. 742–751. DOI: 10.1109/CVPR46437.2021.00080.

[94] Edward H. Lee et al. “LogNet: Energy-efficient neural networks using logarithmic computation”. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (2017), pp. 5900–5904. ISSN: 15206149. DOI: 10.1109/ICASSP.2017.7953288.

[95] Larry Pyeatt and William Ughetta. “Exponent Bit”. In: (2016). URL: https://www.sciencedirect.com/topics/engineering/exponent-bit.

[96] E Edwards. “Floating Point Numbers”. In: (2016). URL: https://www.doc.ic.ac.uk/~eedwards/compsys/float/.

[97] Erick Oberstar. “Fixed-Point Representation Fractional Math Revison 1.2”. In: (Aug. 2007). DOI: 10.13140/RG.2.1.3602.8242.

[98] Yufei Ma et al. “Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)* (2016), pp. 1–8. DOI: 10.1109/FPL.2016.7577356.

[99] Philipp Gysel. “Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks”. In: (2016), pp. 1–8. arXiv: 1605.06402. URL: http://arxiv.org/abs/1605.06402.

[100] Hardik Sharma et al. “From high-level deep neural models to FPGAs”. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49 (2016).

[101] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *CoRR* abs/1603.05279 (2016). arXiv: 1603.05279. URL: http://arxiv.org/abs/1603.05279.

[102] Renzo Andri et al. “YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights”. In: *Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI* 2016-Septe (2016), pp. 236–241. ISSN: 21593477. DOI: 10.1109/ISVLSI.2016.111. arXiv: 1606.05487.

[103] Kota Ando and Kodai Ueyoshi. “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: (2016), pp. 24–25. arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830.

[104] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. “Convolutional Neural Networks using Logarithmic Data Representation”. In: (2016). arXiv: 1603.01025. URL: http://arxiv.org/abs/1603.01025.

[105] Xiao Luo et al. “A Survey on Deep Hashing Methods”. In: (2021). arXiv: 2003.03369.

[106] Simonyan Karen & Zisserman Andrew. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *ICLR 2015* (2015), pp. 1–14. arXiv: arXiv:1409.1556v6.

[107] J.Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015). ISSN: 00034487. DOI: 10.1007/s11263-015-0816-y.

[108] Matthew D. Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks”. In: *In Proceddings of ECCV* (2014), pp. 818–833. ISSN: 16113349. DOI: 10.1007/978-3-319-10590-1_53. arXiv: 1311.2901.

[109] G Hinton, Rumelhart, and Williams. “Learning Representations by back-propagating errors”. In: *Nature 323, 533–536* (1986). ISSN: 02632241. DOI: https://doi.org/10.1038/323533a0.

[110] Y.Bengio Y.LeCun, L.Bottou and P.Haffner. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[111] G.Hinton, S.Osindero, and Y.W Teh. “A fast learning algorithm for deep belief nets”. In: *IEEE Transactions on Neural Networks* 17.6 (2006), pp. 1623–1629. ISSN: 10459227. DOI: 10.1109/TNN.2006.880582.

[112] X.Glorot, A.Bordes, and Y.Bengio. "Deep Sparse Rectifier Neural Networks Xavier". In: *Proceedings of the 14th International Con- ference on Artificial Intelligence and Statistics* 15 (2011), pp. 315–323. ISSN: 15208532. DOI: 10.1002/ecs2.1832.

[113] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[114] Hengshuang Zhao et al. "Pyramid scene parsing network". In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-Janua (2017), pp. 6230–6239. DOI: 10.1109/CVPR.2017.660. arXiv: arXiv:1612.01105v2.

[115] I.J. Goodfellow et al. "Generative Adversarial Nets". In: *NIPS proceeding* (2014). ISSN: 10495258.

[116] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. "FPGA-Based accelerators of deep learning networks for learning and classification: A review". In: *IEEE Access* 7 (2019), pp. 7823–7859. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2890150.

[117] Xiaocong Lian et al. "High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1874–1885. ISSN: 1063-8210. DOI: 10.1109/tvlsi.2019.2913958.

[118] Jiantao Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network • Deep Learning and Convolutional Neural Network – V2 : Brief introduction". In: *FPGA 2015 - 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2016), pp. 26–35. DOI: 10.1145/2847263.2847265.

[119] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations". In: *International Conference on Machine Learning* Icml (2017). arXiv: 1703.03073. URL: http://arxiv.org/abs/1703.03073.

[120] Qinyao He et al. "Effective Quantization Methods for Recurrent Neural Networks". In: (2016), pp. 1–10. arXiv: 1611.10176. URL: http://arxiv.org/abs/1611.10176.

[121] Patrick Judd et al. "Stripes: Bit-serial deep neural network computing". In: *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* 2016-Decem (2016), pp. 1–12. ISSN: 10724451. DOI: 10.1109/MICRO.2016.7783722.

[122] Chen Zhang et al. "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.11 (2019), pp. 2072–2085. ISSN: 19374151. DOI: 10.1109/TCAD.2017.2785257.

[123] Angshuman Parashar et al. "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 27–40. ISSN: 0163-5964. DOI: 10.1145/3140659.3080254.

[124] Micron Technology Inc. "DDR3 SDRAM". In: (2006).

[125] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications". In: *2015 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015* (2016), pp. 418–425. DOI: 10.1109/ICCAD.2015.7372600.

[126] Kung. "Why systolic architectures?" In: *Computer* 15.1 (1982), pp. 37–46. DOI: 10.1109/MC.1982.1653825.

[127] Huimin Li et al. "A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)* (2016), pp. 1–9. DOI: 10.1109/FPL.2016.7577308.

[128] Naveen Suda et al. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks". In: *Proc. ACM FPGA* (2016), pp. 16–25.

[129] Di Wu et al. "A high-performance CNN processor based on FPGA for mobilenets". In: *Proceedings - 29th International Conference on Field-Programmable Logic and Applications, FPL 2019* (2019), pp. 136–143. DOI: 10.1109/FPL.2019.00030.

[130] X. Wang et al. "Convergence of Edge Computing and Deep Learning: A Comprehensive Survey". In: *IEEE Communications Surveys and Tutorials* 22.2 (2020), pp. 869–904.

[131] X. Wang et al. "FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things". In: *IEEE Internet of Things Journal* (2020), pp. 1–1. ISSN: 2327-4662. DOI: 10.1109/JIOT.2020.2976702.

[132] V. Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[133] B. L. Deng et al. "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey". In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[134] S. Wiedemann, K. Müller, and W. Samek. "Compact and Computationally Efficient Representation of Deep Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 31.3 (2020), pp. 772–785.

[135] Yash Bhalgat et al. "LSQ+: Improving Low-Bit Quantization Through Learnable Offsets and Better Initialization". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (June 2020).

[136] Jun Fang et al. "Post-Training Piecewise Linear Quantization for Deep Neural Networks". In: *ECCV* (2020).

[137] Moran Shkolnik et al. "Robust Quantization: One Model to Rule Them All". en. In: *arXiv:2002.07686 [cs, stat]* (June 2020). arXiv: 2002.07686.

[138] Sangil Jung et al. "Learning to Quantize Deep Networks by Optimizing Quantization Intervals With Task Loss". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[139] Ruihao Gong et al. "Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (Oct. 2019).

[140] Zhaohui Yang et al. “Searching for Low-Bit Weights in Quantized Neural Networks”. en. In: *arXiv:2009.08695 [cs]* (Sept. 2020). arXiv: 2009.08695.

[141] Kai Zhong et al. “Towards Lower Bit Multiplication for Convolutional Neural Network Training”. en. In: *arXiv:2006.02804 [cs, stat]* (June 2020). arXiv: 2006.02804.

[142] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *arXiv preprint arXiv:1603.05279* (2016). arXiv: 1603.05279.

[143] Arturo Marban et al. “Learning Sparse & Ternary Neural Networks With Entropy-Constrained Trained Ternarization (EC2T)”. In: *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (June 2020), pp. 3105–3113.

[144] S. Wiedemann et al. “Entropy-Constrained Training of Deep Neural Networks”. In: *2019 International Joint Conference on Neural Networks (IJCNN)* (2019), pp. 1–8.

[145] Paul N. Whatmough et al. “DNN Engine: A 28-nm Timing-Error Tolerant Sparse Deep Neural Network Processor for IoT Applications”. In: *IEEE Journal of Solid-State Circuits* 53.9 (2018), pp. 2722–2731. ISSN: 00189200. DOI: 10.1109/JSSC.2018.2841824.

[146] Dongjoo Shin, Jinmook Lee, and Yoo Hoi-Jun. “DNPU:An 8.1TOPS/W Reconfigurable CNN-RNN Processor for General-Purpose Deep Neural Networks”. In: *ISSCC* (2017). ISSN: 01936530. DOI: 10.1109/ISSCC.2017.7870350.

[147] Hardik Sharma et al. “Bit fusion: Bit-Level dynamically composable architecture for accelerating deep neural networks”. In: *Proceedings - International Symposium on Computer Architecture* (2018), pp. 764–775. ISSN: 10636897. DOI: 10.1109/ISCA.2018.00069. arXiv: 1712.01507.

[148] N. P Jouppi and et.al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *44th International Symposium on Computer Architecture (ISCA)* (2017). ISSN: 00280836. DOI: 10.1038/264079a0.

[149] Jinmook Lee et al. “UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision”. In: *IEEE Journal of Solid-State Circuits* PP (2018), pp. 1–13. ISSN: 0018-9200. DOI: 10.1109/JSSC.2018.2865489.

[150] Song Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016* 16 (2016), pp. 243–254. DOI: 10.1109/ISCA.2016.30. arXiv: arXiv:1602.01528v2.

[151] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks”. In: (2017). arXiv: 1708.04485.

[152] Y. Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[153] Jiecao Yu et al. “Scalpel: Customizing DNN pruning to the underlying hardware parallelism”. In: *Proceedings - International Symposium on Computer Architecture* Part F1286 (2017), pp. 548–560. ISSN: 10636897. DOI: 10.1145/3079856.3080215.

[154] Daniel Bankman et al. “An Always-On 3.8 μj/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS”. In: *IEEE Journal of Solid-State Circuits* 54.1 (2019), pp. 158–172. ISSN: 00189200. DOI: 10.1109/JSSC.2018.2869150.

[155] Angad S. Rekhi et al. “Analog/mixed-signal hardware error modeling for deep learning inference”. In: *Proceedings - Design Automation Conference* (2019), pp. 4–9. ISSN: 0738100X. DOI: 10.1145/3316781.3317770.

[156] Yunzhi Duan et al. “Energy-Efficient Architecture for FPGA-based Deep Convolutional Neural Networks with Binary Weights”. In: *International Conference on Digital Signal Processing, DSP* 2018-Novem (2019), pp. 12–16. DOI: 10.1109/ICDSP.2018.8631596.

[157] Xiaocong Lian et al. “High-performance fpga-based cnn accelerator with block-floating-point arithmetic”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1874–1885. ISSN: 15579999. DOI: 10.1109/TVLSI.2019.2913958.

[158] Shuo Wang et al. “C-Lstm”. In: *FPGA 2018- Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays8- Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2018), pp. 11–20. DOI: 10.1145/3174243.3174253.

[159] Runbin Shi et al. “E-LSTM: Efficient inference of sparse lstm on embedded heterogeneous system”. In: *Proceedings - Design Automation Conference* 5 (2019). ISSN: 0738100X. DOI: 10.1145/3316781.3317813.

[160] M. Panwar et al. “Modified distributed arithmetic based low complexity CNN architecture design methodology”. In: *2017 European Conference on Circuit Theory and Design (ECCTD)* (2017), pp. 1–4. DOI: 10.1109/ECCTD.2017.8093254.

[161] Thomas Wiegand and Heiko Schwarz. “Source Coding: Part I of Fundamentals of Source and Video Coding”. In: *Foundations and Trends® in Signal Processing* 4.1–2 (2011), pp. 1–222. ISSN: 1932-8346. DOI: 10.1561/2000000010.

[162] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *CoRR* abs/1308.3432 (2013). arXiv: 1308.3432.

[163] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[164] Rishikesh Gajjala et al. “Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning”. In: (Dec. 2020). DOI: 10.1145/3426745.3431334.

[165] Yundong Zhang et al. “Hello Edge: Keyword Spotting on Microcontrollers”. In: *arXiv:1711.07128 [cs, eess]* (Feb. 2018). arXiv: 1711.07128.

[166] Bo Liu et al. “An Ultra-Low Power Always-On Keyword Spotting Accelerator Using Quantized Convolutional Neural Network and Voltage-Domain Analog Switching Network-Based Approximate Computing”. en. In: *IEEE Access* 7 (2019), pp. 186456–186469. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2960948.

[167] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. en. In: (Apr. 2009), p. 60.

[168] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010).

[169] Jia Deng et al. “ImageNet: A Large-Scale Hierarchical Image Database”. In: *CVPR09* (2009).

[170] Gianmarco Dinelli et al. “An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick”. In: *International Journal of Reconfigurable Computing* 2019 (2019). ISSN: 16877209. DOI: 10.1155/2019/7218758.

[171] Shouyi Yin et al. “A 1.06-to-5.09 TOPS/W Reconfigurable Hybrid-Neural-Network Processor for Deep Learning Applications”. In: *IEEE Symposium on VLSI Circuits* (2017), pp. C26–C27.

[172] Bo Liu et al. “EERA-ASR: An Energy-Efficient Reconfigurable Architecture for Automatic Speech Recognition with Hybrid DNN and Approximate Computing”. In: *IEEE Access* 6 (2018), pp. 52227–52237. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2870273.

[173] Ruiqi Guo et.al. “A 5.1pJ/Neuron 127.3us/Inference RNN-based Speech Recognition Processor using 16 Computing-in-Memory SRAM Macros in 65nm CMOS”. In: *IEEE Symposium on VLSI Circuits Digest of Technical Papers* (2019), p. 4.