Algorithm and Hardware Design for Efficient Deep Learning Inference

by

Abinash Mohanty

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved October 2018 by the
Graduate Supervisory Committee:

Yu (Kevin) Cao, Chair
Jae-sun Seo
Sarma Vrudhula
Chaitali Chakrabarti

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

Deep learning (DL) has proved itself be one of the most important developements till date with far reaching impacts in numerous fields like robotics, computer vision, surveillance, speech processing, machine translation, finance, etc. They are now widely used for countless applications because of their ability to generalize real world data, robustness to noise in previously unseen data and high inference accuracy. With the ability to learn useful features from raw sensor data, deep learning algorithms have out-performed tradinal AI algorithms and pushed the boundaries of what can be achieved with AI. In this work, we demonstrate the power of deep learning by developing a neural network to automatically detect cough instances from audio recorded in un-constrained environments. For this, 24 hours long recordings from 9 different patients is collected and carefully labeled by medical personel. A pre-processing algorithm is proposed to convert event based cough dataset to a more informative dataset with start and end of coughs and also introduce data augmentation for regularizing the training procedure. The proposed neural network achieves 92.3% leave-one-out accuracy on data captured in real world.

Deep neural networks are composed of multiple layers that are compute-/memory-intensive. This makes it difficult to execute these algorithms real-time with low power consumption using existing general purpose computers. In this work, we propose hardware accelerators for a traditional AI algorithm based on random forest trees and two representative deep convolutional neural networks (AlexNet and VGG). With the proposed acceleration techniques, $\sim 30\times$ performance improvement was achieved compared to CPU for random forest trees. For deep CNNS, we demonstrate that much higher perfance can be achieved with architecture space exploration using any optimization algorithms with system level performance and area models for hardware primitives as inputs and goal of minimizing latency with given resource constraints.

With this method, $\sim$ 30GOPs performance was achieved for Stratix V FPGA boards.

Hardware acceleration of DL algorithms alone is not always the most efficient way and sufficient to achieve desired performance. There is a huge headroom available for performance improvement provided the algorithms are designed keeping in mind the hardware limitations and bottlenecks. This work achieves hardware-software co-optimization for Non-Maximal Suppression (NMS) algorithm. Using the proposed algorithmic changes and hardware architecture

With CMOS scaling coming to an end and increasing memory bandwidth bottlenecks, CMOS based system might not scale enough to accomodate requirements of more complicated and deeper neural networks in future. In this work, we explore RRAM crossbars and arrays as compact, high performing and energy efficient alternative to CMOS accelerators for deep learning training and inference. We propose and implement RRAM periphery read and write circuits and achieved $\sim 3000\times$ performance improvement in online dictionary learning compared to CPU.

This work also examines the realistic RRAM devices and their non-idealities.We do an in-depth study of the effects of RRAM non-idealities on inference accuracy when a pretrained model is mapped to RRAM based accelerators. To mitigate this issue, we propose Random Sparse Adaptation (RSA), a novel scheme aimed at tuning the model to take care of the faults of the RRAM array on which it is mapped. Our proposed method can achieve inference accuracy much higher than what traditional Read-Verify-Write (R-V-W) method could achieve. RSA can also recover lost inference accuracy $100\times \sim 1000\times$ faster compared to R-V-W. Using 32-bit high precision RSA cells, we achieved $\sim 10\%$ higher accuracy using fautly RRAM arrays compared to what can be achieved by mapping a deep network to an 32 level RRAM array with no variations.

To my family.

First and foremost, I would like to like to thank my exceptional advisor, Yu Cao (Kevin), for his steadfast patience, motivation and guidance throughout my doctorate studies. He has been a great source of inspiration for me. Kevin is an extraordinary researcher and taught me the importance of persistance in successful research. I would also like to take this oppurtunity to thank my brilliant co-advisor, Jae-sun Seo. Special thanks to my thesis committee members, Sarma Vrudhula and Chaitali Charkrabarti, for their constructive feedback throughout this thesis.

I have been extremely fortunate to collaborate with a wonderful set of colleagues: Ketul Sutaria, Naveen Suda, Zihan Xu, Ankita Bansal, Xiaocong Du, Zheng Li and Devyani Patra. Special mention goes to Ketul Sutaria, who has been a incredible mentor.

I was very lucky to have a wonderful set of friends throughout my life: Abinash Nanda, Shatadal Mishra, Parth Gupta, Priyam Patel, Anupam Acharya, Biplab Mahapatra, Mohit Mohan Sahu, Sandeep Singh and Summit Dhara. It was conversations with you that made difficult times easy. I cannot acknowledge all by name, but let that not diminish my gratitude for all my friends from NIT-Rourkela, Samsung and ASU.

I have been blessed with a wonderful family. Special thanks to my amazing parents, Arun Mohanty and Sephali Mohanty, my brilliant sisters, Sony Mohanty and Sibangee Mohanty, my brother-in-law, Sailor Siraj, and my loving fiance, Sajan Sahili, for their unconditional love and support. Without their guidance, encouragement and countless sacrifies none of this would have been possible. It is to them I dedicate this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES xi

CHAPTER

1 Introduction 1

2 Introduction to Deep Learning 8

2.1 A Closer Look at Neural Networks 9

2.2 Neural Network Layers 10

2.3 Neural Network Architectures 15

2.4 Datasets 19

2.5 ML Frameworks and Hardware 22

3 Designing a Neural Network 23

3.1 Background and Motivation 23

3.2 Data Collection 25

3.3 Data Preprocessing 28

3.4 Feature Extraction 30

3.5 Neural Network Model for Cough Detection 32

3.6 Results 32

3.7 Conclusion 34

4 Hardware Acceleration using FPGA 36

4.1 High Level Synthesis 36

4.1.1 Altera OpenCL Framework 38

4.1.2 Xilinx HLS Framework 40

4.2 Face Detection using Random Forest Tree 43

4.2.1 Algorithm background 44

CHAPTER Page

4.2.2 FD Accelerator Design ... 44

4.2.3 Results ... 50

4.3 Convolution Neural Networks ... 53

4.3.1 CNN Accelerator Design ... 55

4.3.2 Design Space Exploration ... 61

4.3.3 Optimization Framework ... 67

4.3.4 Results ... 68

4.3.5 Conclusion ... 74

5 Hardware Software Co-Optimization ... 75

5.1 Non-Maximal Suppression ... 76

5.1.1 NMS computation complexity ... 78

5.1.2 Fast and Hardware Efficient NMS ... 80

5.2 Conclusion ... 86

6 Beyond CMOS ... 87

6.1 Introduction ... 87

6.2 Crosspoint Array Architecture and Design ... 91

6.2.1 Read: Integrate and Fire ... 91

6.2.2 Write: Timing based Local Programming ... 93

6.3 65nm CMOS Implementation ... 95

6.3.1 Read ... 96

6.3.2 Write ... 97

6.4 Demonstration in Learning ... 98

6.5 Conclusion ... 99

7 Random Sparse Adaptation ... 100

CHAPTER Page

7.1 Introduction 100

7.2 Non-ideal effects in a RRAM device 103

7.3 Random Sparse Adaptation 104

7.3.1 Regularized random sparse selection 109

7.3.2 Network adaptation using RSA 110

7.4 Demonstration of RSA 111

7.5 Conclusions 114

8 Summary 116

REFERENCES 118

LIST OF TABLES

Table Page

2.1 Properties of several commonly used activation functions. Activation functions are used to introduce non-linearity between layers in deep networks. In the absence of non-linear activation functions, neural networks will essentially be linear mappings from input features to output labels and thus optimization algorithms cannot fit complicated datasets. This makes activation functions critical part of neural network architecture design. ReLU being very optimized for hardware execution is one of the most popular activation functions. 14

3.1 Detailed Participant Information for collection of audio data used for automatic cough detection. The data consists of 24 hours recordings of 9 patients in un-controlled environments. The data is collected using FDA approved VitaloJAK device. 26

3.2 Leave-one-out specificity, sensitivity, and accuracy of proposed algorithm. For this purpose, the network was trained using data from 8 patients and tested on the data from the remaining patient. 33

4.1 Vivado HLS Key Optimization Directives. These directives are added to functionally correct C/C++ codes and then compiled to hardware. Keeping hardware in mind, these directives can be used to derive highly threaded hardware which can be optimized for performance, power and area. This method removes the need to code cycle-to-cycle detailed RTL and thus improves prototyping time drastically. 42

Table Page

4.2 Face detection run time comparison. 4 parallel classifiers operating in parallel with adaptive stride scheme and step size of 0.2, achieves $\sim 30\times$ improvement in speed over CPU with minimal reduction in detection accuracy. 52

4.3 Operations in AlexNet CNN Model (Krizhevsky *et al.* (2012)). To classify a 200×200 input image $\sim$ 1.9GOPs are required. Convolution and fully connected layers dominate in terms of operation requirement. 54

4.4 Comparison of FPGA accelerator boards. 69

4.5 Summary of Execution time and Utilization. 70

4.6 Optimized parameters. 71

4.7 Classification time/image and overall throughput. 71

4.8 Model accuracy comparison. 73

6.1 PARCA operations for key sparse coding tasks. 92

6.2 Evaluation of the speedup in computing and energy. 99

7.1 Assumptions of major types of RRAM device non-idealities. Write variations is considered to follow a normal distribution with mean at the desired value. Stuck-at-high (SF1) arises when certain cells are always at low impedance state no matter what value in written to it. Similarly, Stuck-at-low (SF0) are cells which are always at high impedance state irrespective of the value written to them. 105

7.2 Timing parameters and sizes for RRAM and on-chip memory. On-chip memory, such as Register File (RF), is much faster in Write, but has a larger size. 107

Table | Page

7.3 High cost in operation time when R-V-W is applied. This is due to both the long Write time of RRAM devices and the ineffectiveness of R-V-W, even though in R-V-W the parameters are sorted first by their values and top ones are verified. As observed, for MNIST, to recover inference accuracy within 1% of maximum achievable with 32 level RRAM devices R-V-W needs to correctly program top 40% of the parameters which takes $\sim$ 82 seconds. For CIFAR-10, even with correct programming of 100% of the RRAM cells, we can reach accuracy of 65.18% and it takes $\sim$ 2389 seconds. This shows that, verifying and correctly programming every cell to encode desired conductance values is very in-efficient. 108

LIST OF FIGURES

Figure Page

2.1 Major components of deep neural networks. Feature extraction network learns to extract very high level abstract features which are then flattened and used as input to the fully connected classifier. For example, in convolutional neural networks use convolution operations to take advantage of spatial relation of features in images. The convolution layers learn to extract low level features like edges. They combine edges in succesive layers to create more complicated mid level features like squares, circles etc. They combine mid level features to create high level abstract features like eyes, tyres, lips etc. 10

2.2 Max pooling along a feature map with a 2×2 kernel and stride size of 2. Average pooling operates similarly but does average operation. Global average pooling averages all activations in a given channel to produce one output activation. 11

2.3 Pooling is a very efficient method of removing redundant low level features without removing prominent and winning features. Downsampling caused by pooling layer helps in reducing dimensionality of lower-level features. 12

2.4 Typical hidden layer in multi layer perceptron (MLP) network. This can also be considered as a fully connected layer. In MLP, every output neuron is connected to all input features. Output of each output neuron can be considered as a weighted average of all input features. So for a given input feature vector, the vector representing the values of all output neurons can be obtained by doing a matrix-vector operation. Value of the weight matrix is learned by the training algorithm. 16

2.5 Architecture of LeNet-5 (LeCun *et al.* (1998)). 17

2.6 Architecture of AlexNet (Krizhevsky *et al.* (2012)). 17

2.7 Architecture of VGG-16 (Simonyan and Zisserman (2014)). 18

2.8 Architecture of ResNet (He *et al.* (2016)). 18

2.9 Architecture of GoogleNet (Szegedy *et al.* (2015)). 18

2.10 Architecture of Faster R-CNN (Girshick (2015)). This network performs the feature extraction using convolution layers from classification networks like VGG, AlexNet etc. Apart from that it has a RPN (region proposal network) that produces initial proposals for objects which are pooled to fixed size using ROIPooling. A fully connected classification network uses the pooled high level convolution features to perform classification and fine tuning of bounding boxes. The convolutional feature extraction layers are initialized with weights from any pre-trained classification network, where as the fully-connected layers and RPN is initialized using random normal sampling. The whole network is trained end-to-end with the new dataset. This finetunes the feature extraction convolution layers for the object detection task. 20

2.11 Size normalized examples from MNIST (LeCun *et al.* (2010)). 21

2.12 Random examples from 10 classes of Cifar-10 dataset (Krizhevsky and Hinton (2009)). 22

3.1 Spectrogram of speech and noise compared with three coughs. Cough has distinct high frequency components which are absent in speech and other sounds. 27

Figure | Page

3.2 Preprocessing algorithm extracts a more descriptive cough label. The dataset has event based labeling from medical personnel. Preprocessing first calculates energy in the audio around the labeled cough event. It then detects the maximum energy point. Then it looks for the instance to the left of the peak with energy equal to 15% of the peak energy and labels it as start of the cough. Similarly it looks for instance to the right of the peak with energy equal to 10% of peak energy and labels it as end of the cough. Time duration between start and end is considered as cough duration. Using this method, event based cough dataset is converted to a more informative dataset with start and end of coughs which is then used for training the neural network. 29

3.3 Each 200 ms frame is subdivided into 50 ms windows - 42 (MFCCs, Long bank and delta) features are calculated for each 50 ms segment. 50ms was chosen as the sub-window size as 99.7% of the coughs in the dataset were longer than 50ms. 200ms was used as the frame size as the average cough duration was 181ms. 30

3.4 Proposed network architecture for cough detection. The network consists of 2 hidden layers each with 512 neurons. The input consists of 168 features consisting of MFCCs, Log banks and Deltas from the 4 50ms non-overlapping windows of the audio signal. The network is trained using stochastic gradient descent with L2 weight regularization. 31

Figure Page

3.5 Receiver Operating Characteristic (ROC) of our algorithm averaged across all participants. Larger area under the curve means the algorithm has better performance. Area under the curve (AUC) for the proposed neural network is 0.93 (state-of-the-art at the moment for medical data). 32

4.1 Comparison between CPU, FPGA and GPU for deep learning inference. CPUs are the most programmable and thus are flexible to support all possible algorithms. But the flexibility comes at the cost of poor performance. GPUs on the other hand have highly threaded architecture and thus have the highest performance. They are also programmable to support multiple algorithms. However, GPUs are very power hungry. FPGAs with their programmable fabric are used to create custom hardware for any algorithm and thus have performance much better than CPUs while their power consumption is much smaller compared to GPUs. 37

4.2 Design flow for high level synthesis (HLS). First the C++ codes are made functionally correct with fast emulation mode of HLS. After that HLS directives are used to explore architectural space till we satisfy the throughput and power requirements. C++ emulation and compiling high level codes to RTL ensures fast prototyping and low turnaround time to market. 38

Figure Page

4.3 Design flow of OpenCL based FPGA accelerator. The heterogeneous system consists of two parts: (a) Host CPU running C/C++ codes, (b) FPGA accelerator device programmed with RTL file generated using openCL kernel. The host and the device communicate using PCIe port. The host executes the main task and offloads compute intensive portions of the application to FPGA accelerator. 39

4.4 System diagram of C based HLS accelerator from Xilinx. The host (master) and the accelerator (slave) both sit on the same SoC using AXI bus. The master is responsible for transferring data to the accelerator and programming it. The FPGA accelerator has both AXI master (for high performace DMA transfers from external memory) and AXI slave interfaces (handshake protocol with CPU master). 41

4.5 Architecture of face detection model in Mathias *et al.* (2014). They use 30 difference scales of the original image (for faces of different sizes) and 10000 week classifiers (for faces with different orientations). A total of 10 channels are generated using the scaled image. A scanning window with a rigid template based classifier is used to detect faces. Non Maximal suppression (NMS) is then used to remove redundant detections and preserve the best detections for final box drawing. 45

4.6 Time profiling of face detection algorithm on Intel Core i5-4590 CPU. Computation of 10,000 weak classifiers at all positions on the 30 different scaled versions of the input image is the most time consuming part. It consumes $\sim$ 91% of the total time. So the heterogeneous system was designed so that the FPGA device will accelerate the boosted classifier computation while the CPU handles the rest. 46

4.7 Shift register implementation to store the channel data. This architecture allows loading only the new row while the rest of window is reused from previous iteration, thus saving 95% of the integral data transfer time. 47

4.8 Performance (left) and FPGA resource utilization (right) for different number of parallel compute classifier units. Execution time reduces with increase in number of parallel classifiers till 4, after that it increases because of memory access contention between the parallel execution units. From the resource utilization plot we see that 4 parallel classifiers can be accommodated in the given FPGA device with resource utilization around 50%. 50

4.9 Precision vs recall curves of the CPU+FPGA implementation of the model tested on downsized AFW database using (a) different strides and (b) different scaling factor step sizes. 51

4.10 Mapping 3D convolutions to matrix multiplications. 57

4.11 Accelerating matrix multiplications in OpenCL. 57

4.12 Piece-wise linear approximation of normalization operation kernel with a maximum error of 1%. Using piece-wise lookup tables, normalization is performed without the need for expensive hardware for performing non-linear functions. 60

4.13 Kernel frequency modeling from full synthesis data at 5 random seeds. RMS error of the fit: 12.57 MHz 63

4.14 Run time model vs. measured time of convolution layers 1-5 for a sweep of matrix multiplication block size (N_{CONV}) for SIMD vectorization factor, $S_{CONV} = 1$ and 4. 64

4.15 The execution time model vs. measured data of normalization and fully connected layers in AlexNet for sweep of loop unroll factors N_{NORM} and N_{FC}. 65

4.16 Resource utilization empirical models for normalization block. 66

4.17 Optimization progress of AlexNet implementation. Design variables (N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}) are shown at points A, B and C. 70

4.18 Execution time and resource utilization of each CNN layer type for AlexNet implementation on P395-D8 and DE5-Net FPGA boards. 72

5.1 Top1 vs. operations, size $\propto$ parameters (Canziani *et al.* (2016)). Newer network architectures are much more efficient with respect to model size and the number of operations required. 75

Figure Page

5.2 Distribution of proposal scores for a typical image in Faster-RCNN network. As very few anchors overlap properly with the ground truth objects and majority of the anchors have partial or no overlap, the distribution of the scores is gaussian with mean (peak) at low/negative scores. Top proposals which are necessary for correct detections are in the right tail region of the distribution. These proposals can be easily extracted by estimating mean(μ) and standard deviation(σ) of all scores and then discarding proposals with score smaller than $\mu + \beta \times \sigma$. β is a empirical parameter determined to minimize training dataset. 81

5.3 Proposed structure of internal register array for suppression phase of NMS. The cells are connected in a chained fashion to facilitate efficient ($O(1)$) insertion and deletion of new data. The cells store previous/next/current/new data based on the instruction that is provided to it using the instruction port. 83

6.1 Similarity of biological neural network and the RRAM crosspoint array, in both the network structure and device plasticity. The conductance of RRAM cells can be programmed to particular values using programming voltage of specific pulse width. 88

6.2 PARCA architecture with peripheral Read and Write modules. Z and X (or r) nodes have the same Read (Section 6.2.1), but different Write circuits (Section 6.2.2). All RRAM cells are Read or Written in parallel. 90

6.3 Circuit schematics of the Read circuit. Based on the IF neuron model, it converts a wide range of input current $I_{r,i}$ into a digital number. 93

6.4 Write circuit for Z, with two periods for $r > 0$ and $r < 0$. 94

6.5 Write circuit for r, with the firing rate proportional to r. 95

6.6 The operation of the read circuit for two input current: (left) $I_r = 6.5\mu A$; and (right) $I_r = 1.1\mu A$; the corresponding n_i is 6 and 1. 96

6.7 The overlap in time between Z and r pulses tunes D. 97

6.8 Quantization of read and write circuits are shown. (a) Number of pulses and RRAM current show a close-to-linear relationship. (b) Digitally programmed pulse width closely follows the mathematical multiplication. 98

6.9 Demonstration of dictionary learning with MNIST data. 99

7.1 Each layer in a deep neural network can be mapped to a RRAM array for acceleration. The neural network is first trained offline and the optimized model paramters are selected. The weights of connections in neural network are then encoded as conductance values of RRAM devices using write pulses of appropriate pulse widths. Layerwise execution can be performed in parallel given that we have enough read circuits to collect and digitize column outputs. 102

Figure Page

7.2 The deviation of model parameters after write to the realistic RRAM array. While the pre-trained models are 32-bit floating point numbers, real RRAM devices can have 32 level of quantization. This distorts the distribution of model parameters by forcing pretrained weights to nearest quantization level. Also RRAM cannot encode the value of 0, because the conductance value cannot be 0. As a result all parameters close to 0 are forced to the minimum value that can be encoded by the RRAM (based on the maximum resistance state). That results in a step near values of 0 as shown here. 104

7.3 Accuracy degradation due to device non-idealities in the form of write variations, for two representative convolutional neural networks (LeNet for MNIST dataset and a 9 layered CNN for CIFAR-10). It is assumed that the write variation follows normal distribution. As demonstrated, deeper networks for more complicated tasks are affected to much greater extent. When write variations have $\sigma \geq 1.0$, the outputs from RRAM array are almost random. 105

7.4 Effect of limited quantization levels available in RRAM devices. It is assumed here that the devices have no write variations. As shown, number of levels is critical to inference, especially for complicated tasks and deeper neural networks. In this work we have assumed 32 levels in RRAM devices. 106

7.5 Effect of all non-idealities in RRAM arrays on inference accuracy. The critical non-idealities include device-to-device write variations, quantization errors, stuck-at-faults (SF0 and SF1). The effects of cycle-to-cycle read variations in RRAM devices is negligible compared to others and is not considered in this work. Write variation and quantization have the most significant impact on accuracy. 106

7.6 A large amount of cells needs to be verified, even if their values are ranked. 107

7.7 RSA randomly selects a certain portion of cells (shadowed cells) and re-trains them. Re-training can be considered as a online learning/adaptation procedure which aims at mitigating the effects of array non-idealities. During the training process, the weights stored on the selected cells are adjusted so as to move the networks transform function to a nearby minimal loss point and thus recover from the lost accuracy. 109

7.8 The network structure, design and flow using RF for RSA cells. The pretrained network is first mapped to the RRAM array. The random connections selected for RSA are mapped to RF cell array and initialized with random normal distribution. RSA backpropagates only goes through RF cells. Since we READ/WRITE on RF cells and RRAM is used in read only mode, RSA is very fast. 110

Figure Page

7.9 A small number of 32-bit RSA cells effectively improves the accuracy. As observed, for simple tasks like MNIST, only $3 \sim 5\%$ of total connections in RSA can push the accuracy back to software baseline. An interesting observation here is that, for complicated tasks like CIFAR-10, RSA can push accuracy to much higher values than what can be obtained with a stand alone ideal RRAM array. Since the RSA cells are 32-bit floating point numbers, they mitigate the effects of 32-level quantization to a great extent and thus achieve higher accuracy. 112

7.10 The training of RSA does not require the full dataset to recover the accuracy. With increase in the number of RSA cells, the number of training iterations gets reduced. This behaviour is expected because with few RSA cells, the optimization algorithm (SGD) has to move the values to greater distance to reach the minima as the degrees of freedom available to SGD is less. So it needs more iterations of weight updates. 113

7.11 RSA rapidly recovers the accuracy, achieving $10 - 100\times$ speedup over R-V-W. With writes to RF cells only, RSA removes the need for slow accurate RRAM writes. 114

Chapter 1

INTRODUCTION

Recent years have witnessed groundbreaking achievements in numerous fields of Artificial Intelligence (AI), with Deep Neural Networks (DNNs) delivering performance comparable to or even better than humans. One prime example of this is Google DeepMind's AI (AlphaGo) defeating the best human player in the game of Go (Silver *et al.* (2016)). DNNs have pushed the performance boundaries of numerous AI applications including computer vision (Szegedy *et al.* (2015), He *et al.* (2016)), natural language processing (Young *et al.* (2018)), machine translation (Klein *et al.* (2017)), speech processing (Sotelo *et al.* (2017)), robotics (Levine *et al.* (2018)).

With exceptional performance, robustness to noise and inherent ability to generalize for real world scenarios, DNNs have proved themselves very useful for applications ranging from simple consumer devices like cell phones (Seide *et al.* (2011), Lemley *et al.* (2017), Chen and Xue (2015)), drones (Palossi *et al.* (2018)) and self driving cars (Huval *et al.* (2015), Bojarski *et al.* (2016), Chen *et al.* (2015a)) to much more impacting and difficult tasks like satellite imagery (Hong *et al.* (2017), Tao *et al.* (2016), You *et al.* (2017)), medical devices (Tseng *et al.* (2017), Rajpurkar *et al.* (2017), Kallenberg *et al.* (2016), Avati *et al.* (2017), Kadambi *et al.* (2018)), industrial automation (Kehoe *et al.* (2015), Manyika (2017), Chang *et al.* (2017)) and finance (Heaton *et al.* (2017), Fischer and Krauss (2018), Chong *et al.* (2017), Cavalcante *et al.* (2016)).

This radical change in AI's performance over past few years can be attributed to three primary reasons:

1. Huge publicly available labeled datasets (Deng *et al.* (2009), LeCun *et al.* (2010)) necessary for training

2. Relentless and pioneering efforts to find better network architectures (Iandola *et al.* (2014), Goodfellow *et al.* (2017), Srivastava *et al.* (2014))

3. Highly parallelized and threaded hardware architectures like GPU (Chen *et al.* (2014)), TPU (Jouppi *et al.* (2017)), etc which support training these networks by significantly reducing time required to train these deep networks

For the deep learning research community, improvement in accuracy was the priority for a long time. They primarily focused on improving network's inference accuracy with no concern for the computations needed to achieve that goal. For example, compared to ResNet-34, ResNet-152 (a very popular and top performing neural network used in computer vision applications, He *et al.* (2016)) reduces error rate by a mere 1.97% at the expense of $\sim$ 300% increase in computation cost. As a result, early deep neural networks had huge parameter size (model size) and humongous computation requirement. The hardware research community during this phase was primarily focused on accelerating the newly developed algorithm using the existing architectures. GPUs, Multi-core CPUs and Clusters were the existing viable options. However, the existing architectures were for designed for deep learning algorithms. So researchers started looking for new architectures better suited for DNNs. Works like Suda *et al.* (2016), Ma *et al.* (2017), Zhang *et al.* (2015) achieved excellent hardware performance for these DNNs. Since, these neural networks were not designed keeping speed and hardware in mind, hardware acceleration is not always optimal. For instance, these networks use 32-bit floating point operations which are very inefficient from hardware point of view. Also layers like *softmax*, *local-response-normalization*, *sigmoid* etc involve non-linear functions which are not supported in hardware and thus require huge *look-up-tables.*

DNNs pushed the boundaries of AI's accuracies but at the cost of limited usability.

They could only be executed in servers with high end GPUs. As a result, their usage was limited to applications using cloud computers with high performance computing. Deployment on edge devices was not feasible because to the following reasons: (1) model size was too big to store on embedded platforms, (2) because of the huge computation requirements, executing these algorithms was not feasible within the power envelope of embedded devices running on batteries (Han *et al.* (2015)). In tasks like object detection both accuracy and speed determine the performance of the algorithm. So, there was a need to develop algorithms which are efficient in both accuracy and speed. This need spearheaded the research for next generation of deep learning algorithms which were designed keeping both speed and accuracy in mind. Works like Rastegari *et al.* (2016), Howard *et al.* (2017), Szegedy *et al.* (2015) are aimed at maximizing network performance while reducing the total model size and computation requirement. Gupta *et al.* (2015) did a detailed study on precision requirements of deep learning algorithms and found out that with careful training 13-bit fixed point numbers can give similar inference accuracies as 32-bit floating point numbers while having dramatic improvement hardware performance in terms of speed, power and area.

In the next phase of development, researchers realized that optimizing hardware architectures for deep learning can lead to improvements in performance, but there shall always be a huge headroom for improvement which can be achieved only by developing algorithms keeping in mind the architecture of hardware platform. So algorithms were developed using methods which were hardware friendly. Han *et al.* (2016) implemented a sparse and compressed representation of the existing DNNs using a recursive process of pruning and re-training and accelerated the execution using an efficient sparse network accelerator. All these point to the fact that optimal execution of neural networks on CMOS hardware can be achieved with full stack

development by co-optimizing hardware and software for each other.

Even with all of the optimizations mentioned above, CMOS has physical limitations. With Moore's law coming to an end (Theis and Wong (2017)) devices no longer scale, thereby ending the improvements in area, power and speed that came with transition to successive lower nodes. That led researchers to look for new architectures with multi-core systems to achieve task and instruction level parallelism (Bekkerman *et al.* (2011)). These systems are limited by the level of parallelism that can be achieved for any given algorithm. Also, since the model parameters and internal activations for deep neural networks can require significant memory, storing everything on-chip is not an option. Because of this, internal activations and model parameters are stored on off-chip DRAM and brought into the accelerator in a tile based fashion. As DRAM bandwidth requirements for deep networks are huge, accelerators are bottlenecked by memory bandwidth.

Issues mentioned above created a need in research community to look into new devices and computing architectures that can mitigate the issues associated with CMOS and initiated the next phase of hardware accelerators development for deep learning. Many emerging devices like Resistive RAM (Wong *et al.* (2012)), Phase-change RAM (Raoux *et al.* (2008)), STT-MRAM (Huai (2008)), etc. and more efficient architectures were developed to mitigate bottlenecks associated with von neumann architectures. Mohanty *et al.* (2017) demonstrated that emerging devices can leverage their physical properties to efficiently perform deep neural network operations thereby improving power, performance and area significantly. However, these emerging devices are associated with non-idealities (Mohanty *et al.* (2017)) like quantization error, device-to-device variations, cycle-to-cycle variations, stuck-at-faults, device calibration, etc. which have limited their large scale production and deployment. New approaches to lessen the effects of these non-idealities are active areas of research in

device, architecture and algorithm development communities.

In this work, we demonstrate the need and criticality of hardware-software co-optimization for efficient execution of deep learning. We demonstrate this with implementation and optimization strategies at various stages of DNN algorithm and hardware development. We implement a deep neural network from scratch for automatic cough detection from audio data. With the proposed pre-processing scheme and neural network architecture, we were able to achieve state-of-the-art accuracy for cough detection out-performing methods based on traditional algorithms like PCA. We also implement hardware accelerators for deep convolutional neural networks and random forest trees using FPGAs. With our proposed optimization strategies, we demonstrated high throughput and efficient execution of these. This work also explores emerging architectures like RRAM crossbars and RRAM arrays to mitigate the bottlenecks associated with CMOS based hardware accelerators. Using our proposed architecture $\sim 3000\times$ performance improvements over CPUs has been demonstrated for online learning. This work also examines the realistic RRAM devices and their non-idealities. In this work, we do an in-depth study of the effects of RRAM non-idealities on inference accuracy when a pretrained model is mapped to RRAM based accelerators. To mitigate this issue, we propose Random Sparse Adaptation (RSA), a novel scheme aimed at tuning the model to take care of the faults of the RRAM array on which it is mapped. Our proposed method can achieve inference accuracy much higher than what traditional Read-Verify-Write (R-V-W) method could achieve. RSA can also recover lost inference accuracy $100\times \sim 1000\times$ faster compared to R-V-W.

Rest of the work is organized as follows. Chapter 2 briefly describes some of the key deep learning concepts and introduces some of the commonly used neural network layers like convolution, pooling, fully-connected etc. It also introduces several neural network architectures for image classification and object detection tasks. It also

introduces the datasets, frameworks and hardware platforms used in this work.

Chapter 3 demonstrates the development process of deep neural networks using a test case of automated cough detection in streaming audio data for medical applications. This chapter explores in detail the development for dataset generation starting from collection, labeling, cleanup and feature set selection. It also describes neural network design, training and evaluation. With the approaches described here, state-of-the-art cough detection algorithm was developed with FDA approved vitaloJAK data.

Chapter 4 describes acceleration of AI algorithms on FPGA. It looks into high level synthesis (HLS), a very popular method of hardware RTL generation from C/C++ codes. In particular it looks in details at the system design and HLS procedure using (1) OpenCL Kernels (Altera's approach) and (2) HLS directives (Xilinx's approach). It also describes the FPGA acceleration of AI algorithms using two test cases, (1) high performance face detection using random forest trees (2) throughput optimization of deep CNNs using FPGA acceleration.

Chapter 5 introduces the concepts of hardware software co-optimization. It demonstrates the benefits of co-optimization using a test case of Non Maximal Suppression (NMS) algorithm. The chapter introduces NMS and the bottlenecks and limitations associated with its hardware acceleration (time complexity increases from $O(NlogN)$ to $O(N^2)$). A novel Fast NMS algorithm is proposed which is developed keeping in mind the hardware architecture and data pattern and thus is very suitable for hardware acceleration. A novel hardware register list, capable for performing $O(1)$ insertion and deletion, is also proposed. With the proposed algorithm changes and hardware changes, the chapter demonstrates significant improvement in performance and the worst case time complexity significantly ($O(N)$)

Chapter 6 discusses training and inference of deep neural networks with RRAM

crossbar structures. In this chapter we discuss the necessary read and write circuits needed for this purpose. With the proposed architecture, we demonstrate $/sim 3000\times$ acceleration compared to CPU.

Chapter 7 discuses in detail the non-ideal behavior of emerging analog devices (RRAMs). It gives a quantitative analysis of various non-idealities in RRAM array on inference accuracy of two representative datasets, MNIST and CIFAR-10. This chapter proposes a fundamentally new approach, Random Sparse Adaptation (RSA), to mitigate the impact with high effectiveness and efficiency. Elimination of Write or device-level characterization to recover the accuracy under RRAM non-idealities. RSA achieves 10-100X speedup compared to R-V-W. The hybrid implementation of RSA using a parallel, small, high precision on-chip memory with the main, large, inaccurate RRAM array, enhancing the accuracy by > 10 % for CIFAR-10 using RRAM only.

Chapter 8 concludes this work

Chapter 2

INTRODUCTION TO DEEP LEARNING

Near future will witness large scale deployment of deep learning (DL) algorithms for a wide range of AI applications. DL algorithms are drastically different from standard software algorithms. They are more statistical modeling of complicated real world problems than software engineering. Upon close examination, one will witness DL algorithms comprising of huge matrices of numbers (model parameters) which are multiplied with input vectors. The critical task of DL algorithm development is generating model parameters which result is best possible outcomes with minimal error for hand labeled data. This process is call training the algorithm and is done using statistical optimization algorithms like Stochastic Gradient Descent (SGD), ADAM, Momentum gradient descent, etc. Once the training is over and model parameters are finalized, the algorithm is deployed to generate output with new data (which were not present in the dataset used for training). This is call inference. The goal of training is to maximize inference accuracy by generalizing the model to predict correct output for random unseen inputs.

In this chapter, we will discuss some of the key concepts related to deep learning. We will first discuss the major structural components of a deep neural network. After that we will look at some of the commonly used layers used in high performing neural networks. Then we will briefly discuss some popular and high performing network architectures used in applications like object recognition, object detection, audio processing etc. We will also look into the datasets used in this work and also the frameworks used during training.

Rest of the chapter is organized as follows. Section 2.1 gives a brief introduction to

deep neural networks and the working principles behind them. It also introduces the major structural components of deep networks used in computer vision task. Section 2.2 describes the key layers used in neural networks. Section 2.3 some commonly used neural network architectures. Section 2.4 introduces the publicly available labeled datasets used in this work. Section 2.5 discusses the deep learning frameworks used for training and inference in this work. It also briefly describes the hardware platforms used in this work.

2.1 A Closer Look at Neural Networks

Neural networks are somewhat inspired by the biological functioning of neurons in brain. However, they do not model the biological neurons exactly. Instead, they can be better described as complicated and layered mappings from input to output designed with the sole aim of generalizing and modeling complicated real world tasks, which would be impossible to accomplish with rule based procedural programming. Deep networks use classifiers to map inputs to outputs classes. Inputs to the classifier can be either raw inputs from sensors or they can be a transformed version of the raw inputs. The part of the neural network that does this tranformation of raw sensor inputs is called feature extraction layers. This is shown in Fig. 2.1. Traditionaly feature extraction was done using handcrafted features. This needed a lot of domain knowledge and engineering effort. Deep learning out-performs traditional AI algorithms because it learns the transformation of raw inputs best suited for the task at hand. Deep learning algorithms can learn very abstract high level features which are then used as inputs to the classifier.

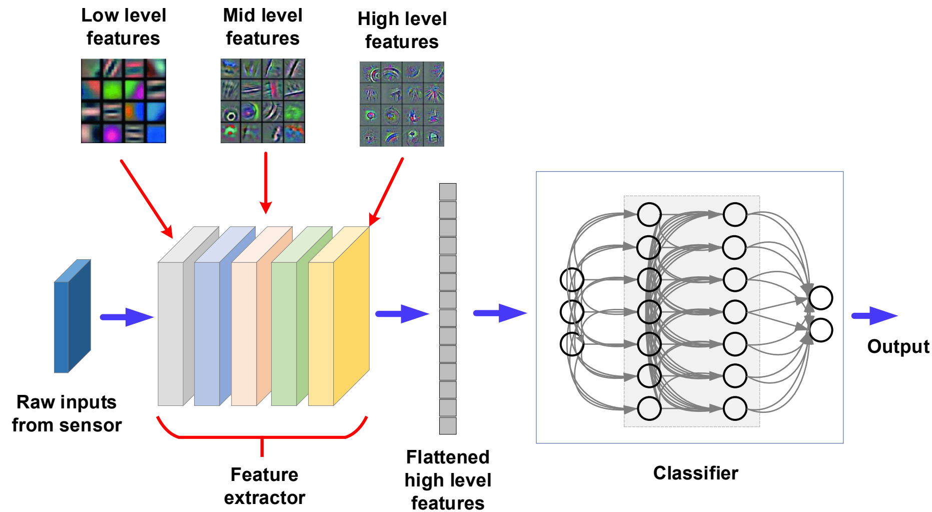


Figure 2.1: Major components of deep neural networks. Feature extraction network learns to extract very high level abstract features which are then flattened and used as input to the fully connected classifier. For example, in convolutional neural networks use convolution operations to take advantage of spatial relation of features in images. The convolution layers learn to extract low level features like edges. They combine edges in succesive layers to create more complicated mid level features like squares, circles etc. They combine mid level features to create high level abstract features like eyes, tyres, lips etc.

2.2 Neural Network Layers

Neural networks have layered structures with current layers feeding data to successive layers. Within each layer, the mappings from input to output is essentially linear. Each layer is generally followed non-linear activation function. It is because of the non-linear activations deep neural networks are able to generalize and model complicated real world datasets. In this section, we will look in details on some the most used layers used in the neural networks.

Convolution

Convolution is the most critical operation of CNNs and it constitutes over 90% of the total operations in AlexNet model (Krizhevsky *et al.* (2012)). It involves 3-dimensional multiply and accumulate operation of N_{if} input features with $K \times K$ convolution filters to get an output feature neuron value as shown in Equation 2.1.

$$out(f_o, x, y) = \sum_{f_i=0}^{N_{if}} \sum_{k_x=0}^{K} \sum_{k_y=0}^{K} wt(f_o, f_i, k_x, k_y) \times in(f_i, x + k_x, y + k_y) \tag{2.1}$$

where $out(f_o, x, y)$ and $in(f_i, x, y)$ represent the neurons at location (x, y) in the feature maps f_o and f_i, respectively and $wt(f_o, f_i, k_x, k_y)$ is the weights at position (k_x, k_y) that gets convolved with input feature map f_i to get the output feature maps f_o.

Normalization

Local Response Normalization (LRN) or normalization implements a form of lateral inhibition (Krizhevsky *et al.* (2012)) by normalizing each neuron value by a factor that depends on the neighboring neurons. LRN across neighboring features and within the

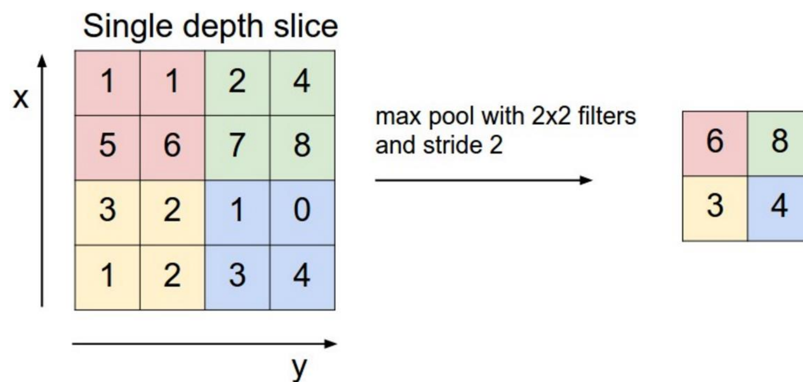


Figure 2.2: Max pooling along a feature map with a 2×2 kernel and stride size of 2. Average pooling operates similarly but does average operation. Global average pooling averages all activations in a given channel to produce one output activation.

same feature can be computed as shown in Equations 2.2 and 2.3, respectively.

$$out(f_o, x, y) = \frac{in(f_o, x, y)}{\left[1 + \frac{\alpha}{K} \sum_{f_i = f_o - K/2}^{f_o + K/2} in^2(f_i, x, y)\right]^{\beta}} \tag{2.2}$$

$$out(f_o, x, y) = \frac{in(f_o, x, y)}{\left(1 + \frac{\alpha}{K^2} \sum_{k_x = x - K/2}^{x + K/2} \sum_{k_y = y - K/2}^{y + K/2} in^2(f_o, x + k_x, y + k_y)\right)^{\beta}} \tag{2.3}$$

Pooling

Spatial pooling or sub-sampling is utilized to reduce the feature dimensions as we traverse deeper into the network. As shown in Equation 2.4, pooling computes the maximum of neighboring $K \times K$ neurons in the same feature map, which also provides a form of translation invariance (Boureau *et al.* (2010)). Although max-pooling is popularly used, average pooling is also used in some CNN models (Boureau *et al.* (2010)). In case of average pooling, we do average operation instead of max of neigh-

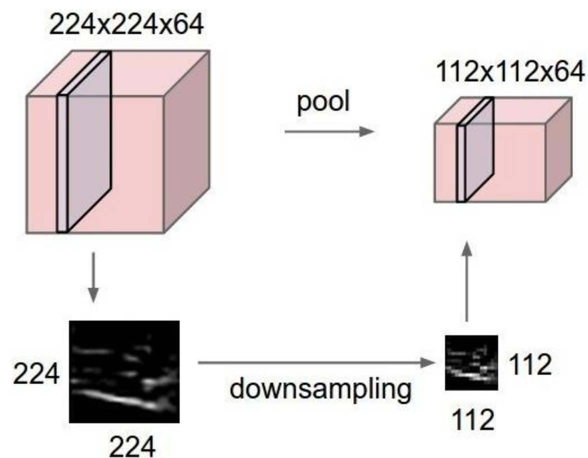


Figure 2.3: Pooling is a very efficient method of removing redundant low level features without removing prominent and winning features. Downsampling caused by pooling layer helps in reducing dimensionality of lower-level features.

boring $K \times K$ neurons in the same feature map. Reducing the dimensionality of lower-level features while preserving the important information, the pooling layer helps abstracting higher-level features without redundancy.

$$out(f_o, x, y) = \max_{0 \leq (k_x, k_y) < K} \big(in(f_o, x + k_x, y + k_y)\big) \tag{2.4}$$

Activation functions

Commonly used activation functions in traditional neural networks are non-linear functions such as tanh and sigmoid, which require longer training time in CNNs. Hence, Rectified Linear Unit, ReLU (Nair and Hinton (2010)), defined as y = max(x,0) has become the popular activation function among CNN models as it converges faster in training. Moreover, ReLU has less computational complexity compared to exponent functions in tanh and sigmoid, also aiding hardware design. PReLU is similar to ReLU except that it has a learnable slope parameter. Softmax and Maxout (Goodfellow *et al.* (2013)) are activation functions that are not functions of a single fold x from the previous layer or layers. Some of the most commonly used activation functions and their properties are listed in Table 2.1.

Fully connected layer

Fully-connected layer or inner product layer is the classification layer where all the input features (N_{if}) are connected to all of the output features (N_{of}) through synaptic weights (wt). Each output neuron is the weighted summation of all the input neurons as shown in Equation 2.5.

$$out(f_o) = \sum_{f_i=0}^{N_{if}} wt(f_o, f_i) \times in(f_i) \tag{2.5}$$

The outputs of the inner-product layer generally traverse through some non-linear activation function to the next inner-product layer or directly to a Softmax function

Table 2.1: Properties of several commonly used activation functions. Activation functions are used to introduce non-linearity between layers in deep networks. In the absence of non-linear activation functions, neural networks will essentially be linear mappings from input features to output labels and thus optimization algorithms cannot fit complicated datasets. This makes activation functions critical part of neural network architecture design. ReLU being very optimized for hardware execution is one of the most popular activation functions.

Name	**Equation**	**Derivative (wrt. x)**
Logistic/Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH	$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Leaky ReLU	$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
PReLU	$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$ for i = 1, ..., J	$\frac{\partial f_i(\vec{x})}{\partial x_j} = f_i(\vec{x})(\delta_{ij} - f_j(\vec{x}))$
Maxout	$f(\vec{x}) = \max_i x_i$	$\frac{\partial f}{\partial x_j} = \begin{cases} 1 & \text{for } j = \operatorname*{argmax}_i x_i \\ 0 & \text{for } j \neq \operatorname*{argmax}_i x_i \end{cases}$

that converts them to probability in the range (0, 1). The final accuracy layer compares the labels of the top probabilities from softmax layer with the actual label and gives the accuracy of the CNN model.

2.3 Neural Network Architectures

In this section, we shall take a look at some of the most common neural network architectures like multi-layer perceptron (MLP), convolutional neural network (CNN), faster RCNN, single shot detector (SSD), recurrent neural network (RNN) etc.

Multi-layer perceptron (MLP)

MLPs have deep layered structures, with each layer feeding data to subsequent layers. Fig. 2.4 illustrates a very simple fully connected feed forward network. Its fully connected because each node (neuron) in any layer is connected to every node in the next layer. Its feed forward because the data movement is always in one direction (input output) and there is no feed-back path. Each layer in the neural network is essentially doing a matrix–vector multiplication on the input data. MLPs are generally used as final classifier layers in complex neural networks and the inputs to MLP are generally features extracted from the the raw input data using the previous layers. In case of CNNs, the convolution layers provide the input features to the MLP for classification. Since the dimension of the inputs to MLPs can be big, MLP generally account for a huge portion of workload when executing a neural network as shown in Jouppi *et al.* (2017). In Xu *et al.* (2017) MLPs have been successfully implemented to do feed forward inhibition for character recognition.

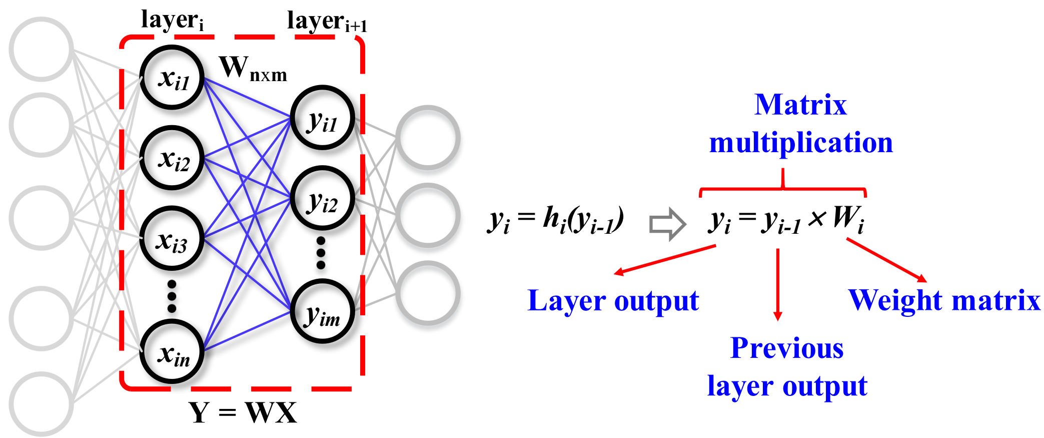


Figure 2.4: Typical hidden layer in multi layer perceptron (MLP) network. This can also be considered as a fully connected layer. In MLP, every output neuron is connected to all input features. Output of each output neuron can be considered as a weighted average of all input features. So for a given input feature vector, the vector representing the values of all output neurons can be obtained by doing a matrix-vector operation. Value of the weight matrix is learned by the training algorithm.

Convolutional neural networks

Convolutional neural networks have been widely used for image based computer vision applications. They take the advantage of spatial locality of images and share the weights in space, thereby making them invariant to translation of the input. Such weight sharing makes the number of weight much smaller compared to fully-connected layer with the same input/output dimensions. As discussed previously, CNNs use trainable convolution filters to extract learned features from input images. Using successive convolutions very high level features are extracted, which are then used for classification using a fully connected classification network. Following are some of the most commonly used CNNs. We used many of these to benchmark some of our works.

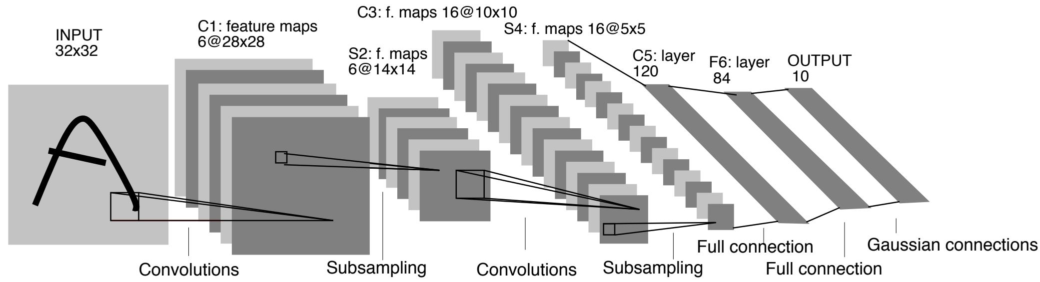


Figure 2.5: Architecture of LeNet-5 (LeCun *et al.* (1998)).

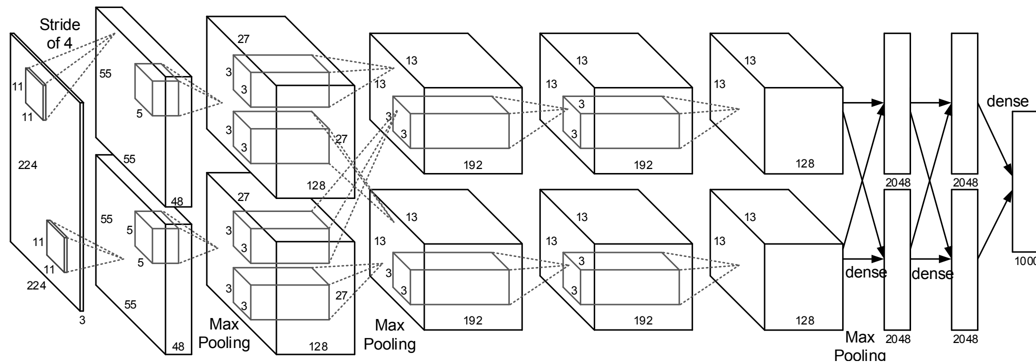


Figure 2.6: Architecture of AlexNet (Krizhevsky *et al.* (2012)).

LeNet-5 (LeCun *et al.* (1990) is one of the first networks which demonstrated that CNNs, with their learnable feature extraction layers, can out perform traditional networks with handcrafted features like HOG, LUV, gradient magnitude etc. It consists of 2 convolution layers and two fully connected layers. It was designed for hand written digits recognition. AlexNet (Krizhevsky *et al.* (2012)) was one of the first works which successfully trained a deep convolutional network. It had 5 convolution layers and 3 fully connected layers with 61 million parameters. The architecture used 3 different types of kernels (3×3, 5×5 and 11×11) and achieved a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3% on ImageNet.

VGG-16 (Simonyan and Zisserman (2014)) is a much larger network compared to LeNet and AlexNet. It features 13 convolutional layers and 3 fully connected layers. This network has a huge number of parameters, 138 million. As the network is deeper,

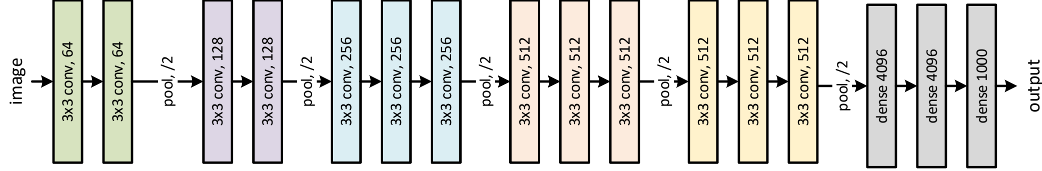


Figure 2.7: Architecture of VGG-16 (Simonyan and Zisserman (2014)).

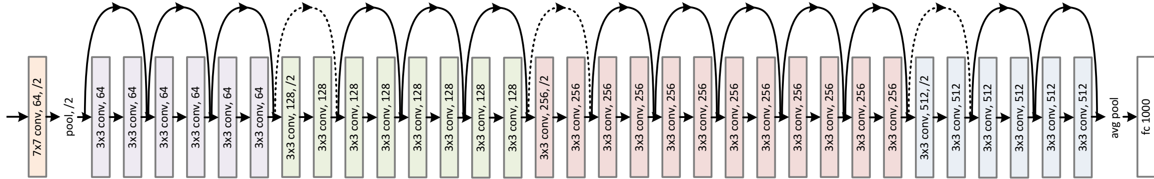


Figure 2.8: Architecture of ResNet (He *et al.* (2016)).

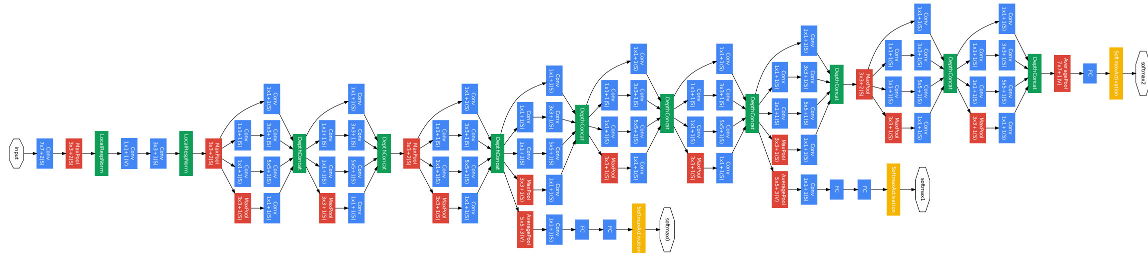


Figure 2.9: Architecture of GoogleNet (Szegedy *et al.* (2015)).

its accuracy is better compared to AlexNet. VGG-16 achieved a top-1 accuracy of 68.5% and a top-5 accuracy of 88.7% on ImageNet. ImageNet pre-trained model for this network is widely used in many computer vision tasks like image classification, detection and segmentation.

ResNet (He *et al.* (2016)) solved the issue of vanising gradients in deeper layers with the residual block with bypass layer. The most popular ResNet (ResNet-50) has 25.5 million parameters. It has 49 convolution layers and 1 fully connected layer. It uses element-wise additions in residual blocks. It achieved a top-1 accuracy of 76.1% and a top-5 accuracy of 92.9% on ImageNet.

GoogleNet (Szegedy *et al.* (2015)) features 9 inception modules each consisting of

4 branches of 1×1, 3×3 and 5×5 convolutions and down-sampling. In total it has 57 convolution layers and 1 fully connected layer. GoogleNet has a total of 7 million parameters. It achieved a top-1 accuracy of 68.9% and a top-5 accuracy of 89.0% on ImageNet.

All the network architectures discussed previously as for image classification task. Object detection on the other hand uses slightly different network architectures. This is because object detection algorithms are required to not only classify all the objects present in the image but also localize them. Faster-RCNN (Girshick (2015)) introduced Region Proposal Network (RPN) that shared full-image convolutional features with the classification network (fc layers), thus enabling nearly cost-free region proposals. RPNs are trained end-to-end to generate high quality region proposals, that are pooled using ROI-Pooling layer and feed into classification network for detection. They used VGG-16 for convolution feature extraction and obtained 5fps on GPUs with object detection accuracy on PASCAL VOC 2007 (73.2% mAP) and 2012 (70.4% mAP) using 300 proposals per image.

2.4 Datasets

Large labelled datasets are cornerstones of deep learning. They are used to train and test the performance of models. We used a variety of tasks through out this work. The datasets used in the experiments for this work include MNIST, Cifar-10, ImageNet, AFW database. We also created an audio dataset of cough recordings that we used for training a MLP. Details about this dataset creation is explained in the next chapter.

MNIST is a dataset for handwritten digits (LeCun *et al.* (2010)) with 60,000 training images and 10,000 test images. There are ten classes with ten digits, and the image size is 2828. Each image is grayscale. This dataset is relatively easy and

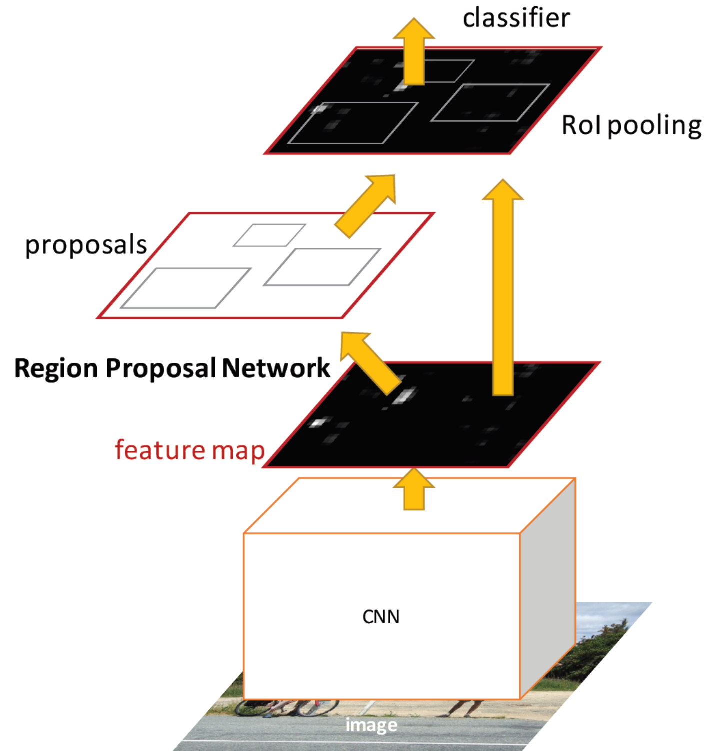


Figure 2.10: Architecture of Faster R-CNN (Girshick (2015)). This network performs the feature extraction using convolution layers from classification networks like VGG, AlexNet etc. Apart from that it has a RPN (region proposal network) that produces initial proposals for objects which are pooled to fixed size using ROIPooling. A fully connected classification network uses the pooled high level convolution features to perform classification and fine tuning of bounding boxes. The convolutional feature extraction layers are initialized with weights from any pre-trained classification network, where as the fully-connected layers and RPN is initialized using random normal sampling. The whole network is trained end-to-end with the new dataset. This finetunes the feature extraction convolution layers for the object detection task.

small, taking the model only a few minutes to train.

Cifar-10 is a dataset of color images (Krizhevsky and Hinton (2009)). It has 50,000 training images and 10,000 test images. There are ten classes, and the image size is

3232. The dataset is slightly more difficult than MNIST but the model still only required a few hours to train. We used Cifar-10 for ablation studies when we needed to repeat a group of similar experiments many times.

AFW dataset (Zhu and Ramanan (2012)) is built using Flickr images. It has 205 images with 473 labeled faces. For each face, annotations include a rectangular bounding box, 6 landmarks and the pose angles.

ImageNet is a large-scale dataset for ILSVRC challenge (Deng *et al.* (2009)). The training dataset contains 1000 categories and 1.2 million images. The validation dataset contains 50,000 images, 50 per class. The classification performance is reported using Top-1 and Top-5 accuracy. Top-1 accuracy measures the proportion of correctly-labeled images. If one of the five labels with the largest probability is a correct label, then this image is considered to have a correct label for Top-5 accuracy. We used the ImageNet dataset to measure the performance of model compression and regularization.

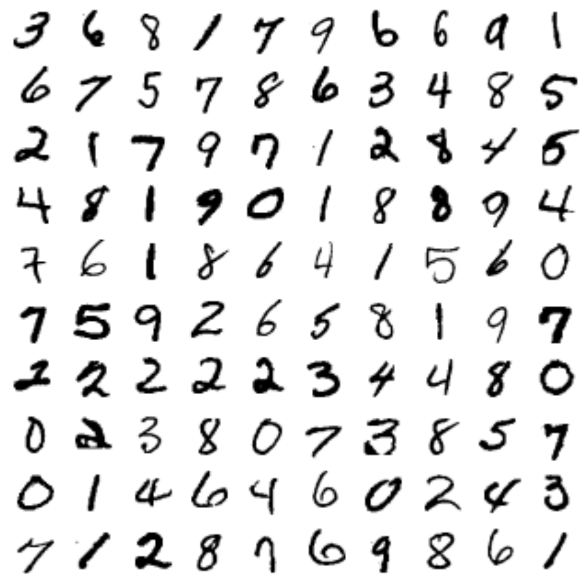


Figure 2.11: Size normalized examples from MNIST (LeCun *et al.* (2010)).

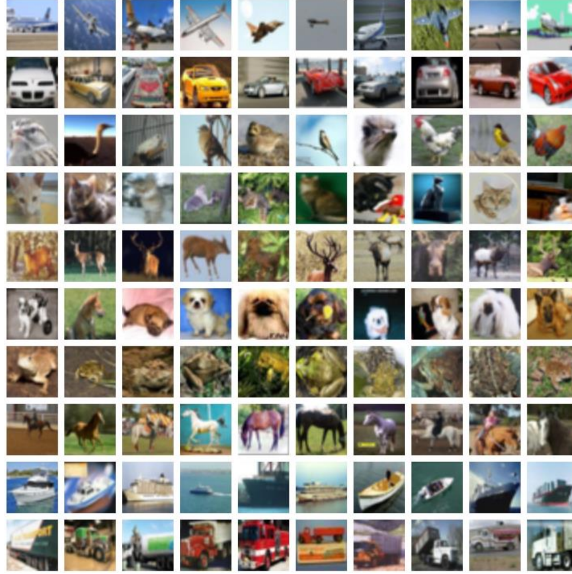


Figure 2.12: Random examples from 10 classes of Cifar-10 dataset (Krizhevsky and Hinton (2009)).

2.5 ML Frameworks and Hardware

In this work, we use Tensorflow (Abadi *et al.* (2016)) for training and evaluating neural networks. Tensorflow is a deep learning framework from Google. All the neural network computations are abstracted to graph operations like convolution, pooling, deconvolution etc. This enables the users to focus on the high level network design. The hardware used for training in this work was a CPU with Core i7-5930K 6-core 3.5GHz desktop processor and NVIDIA Maxwell TitanX GPU with 12GB of memory. For custom hardware accelerators using FPGA, we use DE5-Net FPGA Development Kit with Altera Stratix-V GXA7 FPGA and P395-D8 board with Stratix-V GSD8 FPGA board.

Chapter 3

DESIGNING A NEURAL NETWORK

In the previous chapter, we discussed some of the basics of neural networks. We also looked at a few popular and widely used network architectures. In this chapter, we will discuss in detail the process of designing and training a neural network. For this purpose, we will use development of an automatic cough detection algorithm as the test case. The proposed algorithm was developed to be used in medical devices and analyzes audio files and determines the number of coughs in it. The rest of the chapter is organized as follows. First we shall look at the background of the problem statement and also the motivation behind this work. Then we shall look at how the dataset is collected and designed. Then we shall look at feature extraction from raw data, network architecture definition, training the network and evaluating it with test data.

3.1 Background and Motivation

Coughing is one of the most important and frequent symptoms reported by patients (Ly *et al.* (1999), Gibson *et al.* (2010)). Chronic cough can result in deleterious effects on health and quality of life (Young and Smith (2010)). Monitoring cough symptoms is important in detecting and treating respiratory conditions such as chronic obstructive pulmonary disease (COPD), asthma, pulmonary fibrosis, and tuberculosis (Tracey *et al.* (2011), Kerem *et al.* (2008), Marsden *et al.* (2008)).

To assess the frequency and severity of cough, several subjective tests have been developed (e.g. Leicester cough questionnaire (Birring *et al.* (2003)), visual analog scales, etc.). These methods provide insight into the perceived severity of cough symp-

toms, but are ultimately unreliable when compared to objective methods of studying cough, because factors such as patient mood, vigilance, and the placebo effect can impact the patient's report of cough frequency (Smith and Woodcock (2008), Decalmer *et al.* (2006)).

However, objective tools for studying cough are lacking. One quantitative method to assess cough frequency and severity consists of using ambulatory systems to record audio from patients for an extended time, and then manually counting the number of coughs in the recorded audio. Manually counting coughs is a time-consuming process that requires an expert to verify labeled coughs (Barton *et al.* (2012)). This is impractical for large amounts of data.

A number of automatic cough detection systems have been proposed in the literature. The Leicester Cough Monitor (LCM) Birring *et al.* (2008), and VitaloJAK McGuinness *et al.* (2012) are examples of ambulatory systems consisting of both wearable devices to record patient audio, and algorithms for cough detection from recorded data. The LCM applies a Hidden Markov Model (HMM) trained on mel-frequency cepstral coefficients (MFCCs) in order to detect cough sounds. However, the LCM algorithm is only semi-automated - it requires manual tuning of model parameters for each individual recording (McGuinness *et al.* (2008)). This algorithm takes a 24-hour patient audio recording and creates a shorter recording with all suspected coughs. To decrease false alarm rate, a portion of the detected coughs must be manually confirmed by personnel (Birring *et al.* (2008)).

Recently, there has been significant interest in applying deep learning techniques to automatic cough detection. In Barry *et al.* (2006) the authors devise a probabilistic neural network trained using linear predictive cepstral coefficients (LPCCs) and MFCCs to distinguish cough sounds from background. The authors in Amoh and Odame (2015) and Amoh and Odame (2016) applied convolutional neural networks

(CNNs) trained directly on the short-time Fourier transform (STFT) of audio segments. However, most of these methods are validated on limited datasets collected in artificial environments, or use proprietary hardware for collecting patient data. For example, in Swarnkar *et al.* (2013) the data only consists of three patients recorded in a hospital setting; and, only one patient was recorded for more than four hours. The authors in Amoh and Odame (2015) and Amoh and Odame (2016) use custom hardware to record healthy volunteers reading passages and voluntarily coughing in a controlled lab setting. It is well-known in the literature that voluntary coughs have different patterns from reflex coughs Smith *et al.* (2006). In Liu *et al.* (2015b) a pre-trained neural network is applied to 24-hour patient recordings collected in a restricted, hospital environment. They use custom recording devices rather than using an FDA-cleared cough monitor.

In this work, we propose a framework for audio-based automatic cough detection. The main contributions of this work are: (1) an extensive dataset containing 9 days of audio recorded in real-world conditions, from 9 patients with a variety of respiratory illnesses, using the FDA-cleared cough monitoring device, VitaloJAK; (2) a pre-processing algorithm to fine tune data labels to improve neural network accuracy and convert event based cough labeling to labels containing cough start and end points; (3) a deep neural network (DNN) trained using MFCCs and other features to discriminate cough sounds from background noise. The proposed framework achieves an average leave-one-out cross-validation specificity, sensitivity, and accuracy of 93.7%, 97.6% and 92.3% respectively.

3.2 Data Collection

As mentioned previously, designing a neural network starts with labelled datasets. For this work, we created a custom dataset. Recordings were supplied from an acous-

Participant	Disease	Coughs	Gender
1	Chronic Cough	3133	M
2	Chronic Cough	509	F
3	Chronic Cough	546	F
4	COPD	102	F
5	COPD	852	F
6	Asthma	221	F
7	Asthma	118	M
8	Lung Cancer	163	F
9	Lung Cancer	26	M

Table 3.1: Detailed Participant Information for collection of audio data used for automatic cough detection. The data consists of 24 hours recordings of 9 patients in un-controlled environments. The data is collected using FDA approved VitaloJAK device.

tic cough recording repository (RADAR) maintained at the University Hospital of South Manchester, with patient consent. Sound recordings were collected using the VitaloJAK cough recording device over 24-hour periods; recordings were commenced in a research clinic and then patients were permitted to go about their normal daily routines. The monitors were collected once the recordings were completed. The device makes continuous sound recordings at 8 kHz sample rate, from an air-coupled contact microphone placed over the manubrium sterni and a free-field lapel microphone. We use audio from the lapel microphone for our analysis. Participants were instructed not to remove the device or microphones during the recording and to keep the equipment dry. A total of 9 recordings (3 chronic cough, 2 asthma, 2 chronic

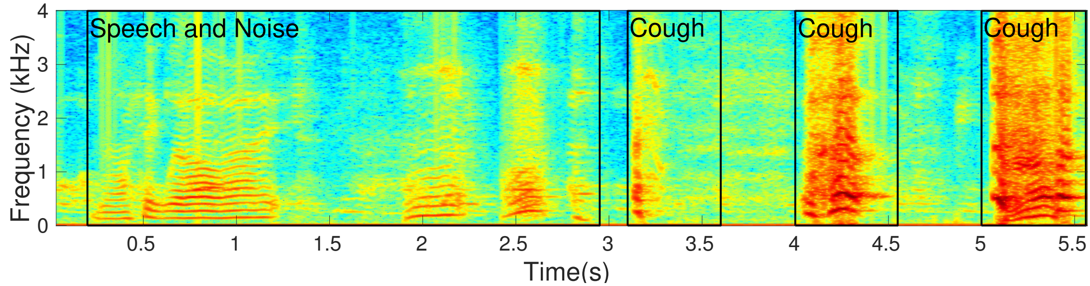


Figure 3.1: Spectrogram of speech and noise compared with three coughs. Cough has distinct high frequency components which are absent in speech and other sounds.

obstructive pulmonary disease and 2 lung cancer patients) were included in the analysis. Each 24 hour recording was listened to in its entirety by technical staff trained in cough identification, and the location of each cough sound heard was recorded electronically.

Our dataset consists of a total of 5,670 coughs. The dataset contains a rich variety of background noise such as music, conversation, watching television, and riding in a car. The audio contains many sounds easily confused with coughing such as throat clearing, sneezing, and laughing. Table 3.1 shows the breakdown of the dataset by disease, cough count, and gender.

Figure 3.1, shows the spectrogram of three coughs of different length and a set of non-cough sounds taken from the dataset. We can make two important observations from this spectrogram. We note that coughing contains a larger amount of energy in higher frequencies than speech or other types of noise. A properly trained DNN can discriminate coughing from background by utilizing these characteristics unique to coughing. Also, any algorithm trained to detect these coughs must be able to account for the variability in cough length and intensity (see Figure 3.1).

3.3 Data Preprocessing

Every cough in the database was manually labeled by a trained expert. The top graph in Figure 3.2 shows how coughs were labeled. As we have noted, coughs vary in duration. However, the provided labels do not reflect this information. If features are extracted from a constant window around the provided labels, background audio events adjacent to coughing can be unintentionally included as part of the cough. Therefore, we must determine the cough start and end times from the provided labels.

The cough reflex consists of three audible portions: 1) a rapid, explosive phase, 2) an intermediate, decaying phase consistent with forced expiration, and 3) a voiced phase (not necessarily present in all coughs). Since the first two phases are ubiquitous across all coughs, they allow us to determine the start and end of a cough using an energy-based criteria.

Figure 3.2 outlines the label preprocessing algorithm. Given the event-based label, we extract a 420 ms window of audio from 70 ms before to 350 ms after the provided label. We choose a window of 420 ms because more than 95% of all coughs in our dataset were observed to be shorter than 400 ms in duration.

Next, an energy versus time profile is generated for the cough. We calculate the energy for every 10 ms frame within the 420 ms window using a step size of 2 ms. We calculate the energy for each 10 ms frame and find the maximum value of this energy profile within the 420 ms window. The first 10 ms frame that precedes the maximum energy frame with 15% of the energy of the maximum energy frame is chosen as the start of the cough; and the first frame that occurs after the maximum energy frame with 10% of the energy of the maximum energy frame is selected as the end of the cough. Any cough found to have a duration of less than 40 ms is pruned from the

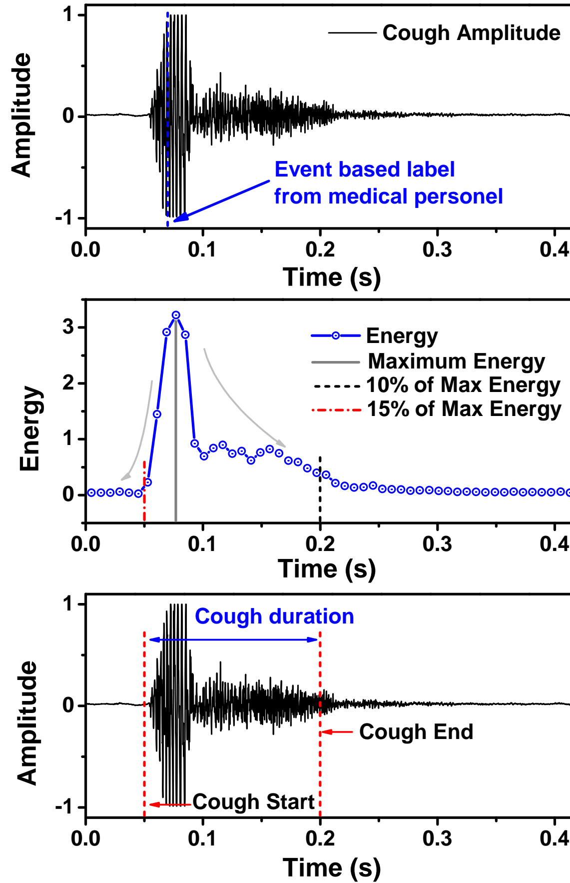


Figure 3.2: Preprocessing algorithm extracts a more descriptive cough label. The dataset has event based labeling from medical personnel. Preprocessing first calculates energy in the audio around the labeled cough event. It then detects the maximum energy point. Then it looks for the instance to the left of the peak with energy equal to 15% of the peak energy and labels it as start of the cough. Similarly it looks for instance to the right of the peak with energy equal to 10% of peak energy and labels it as end of the cough. Time duration between start and end is considered as cough duration. Using this method, event based cough dataset is converted to a more informative dataset with start and end of coughs which is then used for training the neural network.

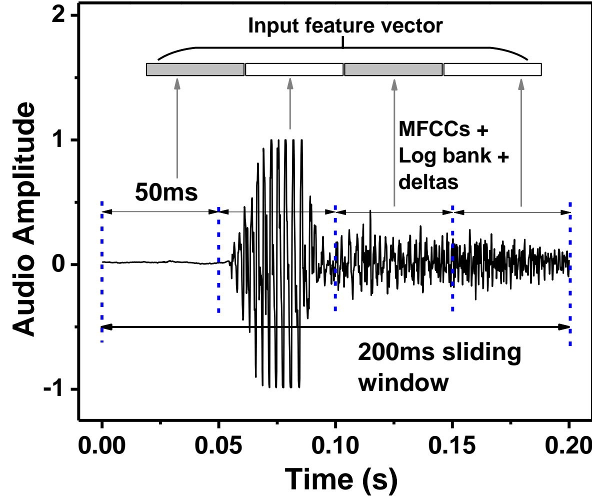


Figure 3.3: Each 200 ms frame is subdivided into 50 ms windows - 42 (MFCCs, Long bank and delta) features are calculated for each 50 ms segment. 50ms was chosen as the sub-window size as 99.7% of the coughs in the dataset were longer than 50ms. 200ms was used as the frame size as the average cough duration was 181ms.

dataset [1] . The resulting dataset consists of coughs ranging from 40 ms to 420 ms duration, with an average duration of 200 ms.

3.4 Feature Extraction

A total of 168 features are used as inputs to the DNN. Since we aim to apply the DNN in a real-time setting in subsequent work, we perform training and inference using 200 ms frames of audio (200 ms corresponds to the average cough length). Four 200 ms training examples are generated from each cough by varying the location of the cough within each training example. This ensures the DNN is invariant to the position of a cough within the frame.

For each cough, two training examples are generated such that the beginning of

[1] Less than 0.1% of all coughs were pruned

the training example can occur anywhere within a 25 ms window before or after the cough start time (with uniform random probability). The remaining two out of four training examples are similarly generated, but the window is increased to 60 ms before or after the cough start.

Then, each 200 ms frame is further subdivided into four 50 ms windows to capture the temporal profile of each audio frame (Figure 3.3). From each 50 ms window we compute 42 features: 13 MFCCs, 13 MFCC delta features, and 13 MFCC delta-delta features. The remaining three features are the log energy within the 13 MFCCs, 13 MFCC delta features, and 13 MFCC delta-delta features. Since we break down each 200 ms frame into four 50 ms windows, we supply our network with 168 input features. To generate non-cough training examples, we randomly sample 200 ms segments of non-cough audio and calculate the same 168 features.

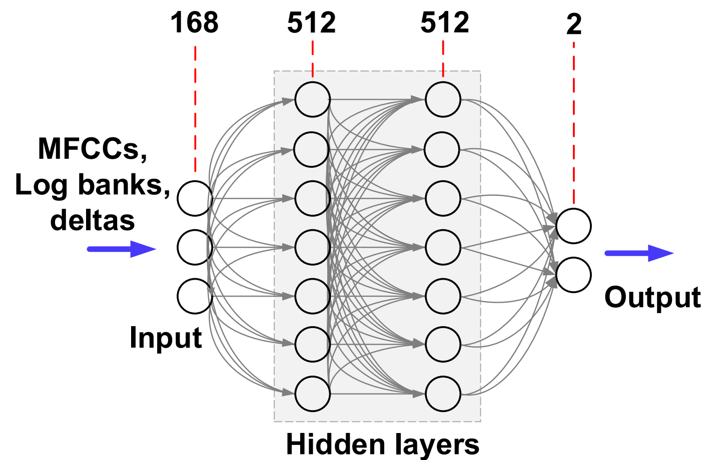


Figure 3.4: Proposed network architecture for cough detection. The network consists of 2 hidden layers each with 512 neurons. The input consists of 168 features consisting of MFCCs, Log banks and Deltas from the 4 50ms non-overlapping windows of the audio signal. The network is trained using stochastic gradient descent with L2 weight regularization.

3.5 Neural Network Model for Cough Detection

Figure 3.4 shows our proposed neural network architecture. The DNN was trained with an equal number of positive and negative examples using stochastic gradient descent (SGD) and momentum. Using a grid search and cross-validation, we employed a learning rate of 0.15, momentum of 0.9, and a batch size of 150. The network was trained for 50 epochs.

3.6 Results

Table 3.2 summarizes the leave-one-out specificity, sensitivity, and accuracy our algorithm achieves. The DNN is trained on all subjects except for one, which was left for testing. This is then repeated across all subjects in the entire dataset (leave-one-participant-out cross validation).

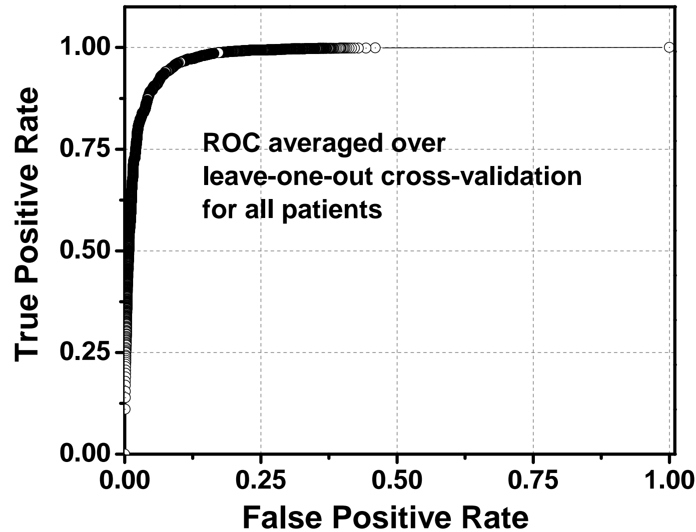


Figure 3.5: Receiver Operating Characteristic (ROC) of our algorithm averaged across all participants. Larger area under the curve means the algorithm has better performance. Area under the curve (AUC) for the proposed neural network is 0.93 (state-of-the-art at the moment for medical data).

Subject	Specificity %	Sensitivity %	Accuracy %
1	97.7	92.2	95.2
2	95	97.4	95.4
3	88.1	97.8	91.9
4	87	97.3	91.2
5	97	97	96.2
6	98.3	97.6	94.3
7	87	94.4	89.7
8	96	96.1	92.5
9	97.3	97.6	92.3
Avg	**93.7**	**97.6**	**92.3**

Table 3.2: Leave-one-out specificity, sensitivity, and accuracy of proposed algorithm. For this purpose, the network was trained using data from 8 patients and tested on the data from the remaining patient.

To compute accuracy, a test data set of four positive examples are created for each cough in accordance with the method outlined in Section 3.4. An equal number of negative test examples are randomly selected from background audio. To calculate sensitivity and specificity, a 200 ms sliding window is extracted from each 24 hour recording, with a step size of 50 ms. We define a false positive as any 200 ms frame that was incorrectly classified as a cough, and was not within 1 second of a cough.

The DNN achieves an average leave-one-out accuracy of 92.3%, with the highest accuracy of 96.2% for participant 5, and lowest accuracy of 89.7% for participant 7. This is due to the unique cough signature of participant 7 which is perceptually similar to throat clearing.

Sensitivity (true positive rate) and specificity (true negative rate) are more useful in describing the DNN's performance on 24 hour segments of audio due to the severely imbalanced classes. The algorithm results in an average specificity and sensitivity of 93.7% and 97.6% respectively.

Participant 4 and 7 both had the lowest specificity of 87%. This is due to the large amount of loud conversation in both of these recordings. Loud speech is one of the most likely sources of false positives. As several authors note, the most difficult part in designing ambulatory cough detection systems is robustness to false alarms. Since classes are heavily imbalanced, specificity must be as high as possible to avoid large numbers of false positives.

As was expected, the sensitivity for participant 7 was lower than average (94.4%) due to the participant's uncommon cough pattern. The lowest leave-one-out sensitivity of 92.2% is found for participant 1. Since 3,133 out of the 5,670 (more than 55% of all training set coughs) come from participant 1, the DNN is likely to have an incomplete model of cough when the recording from participant 1 is left out from training.

The receiver operating characteristic (ROC) averaged across all participants is shown in Figure 3.5. We use the same test data set that is used to find accuracy in creating the ROC. The area under the curve (AUC) of the ROC is 0.93, a value close to 1 indicating that our model performs well in discriminating cough from background.

3.7 Conclusion

In this chapter, we demonstrated that with careful dataset design, data augmentation, network architecture and training scheme, deep neural networks can perform extremely well. We discussed the implementation details of a deep neural network and a data pre-processing algorithms for cough detection from ambulatory data collected

with the FDA-cleared VitaloJAK device. We trained a DNN with two hidden layers on MFCC features to successfully discriminate coughing sounds from background noise. Results indicate our algorithm achieves high sensitivity, specificity, and accuracy on our extensive dataset. The proposed framework could decrease the load on medical personnel in labeling coughs from ambulatory audio recordings.

Chapter 4

HARDWARE ACCELERATION USING FPGA

In the previous chapter, we saw how to design and train a neural network. These algorithms are very compute intensive, thereby making inference on general purpose CPUs extremely slow and inefficient. As shown in Fig. 4.1, GPUs with there highly parallel architecture have proved to be very efficient for deep learning applications. However, inference on GPUs is power hungry. FPGAs on the other hand, with their programmable fabric are very suitable for creating multi-threaded hardware which can be much faster than CPUs while being much more power efficient than GPUs.

In this chapter, we will look at accelerating AI applications using FPGA. First we will take a look at High Level Synthesis framework for design and implementation of accelerators. We will study the two approaches to HLS namely, OpenCL kernels and HLS constructs. Then we shall take two design examples of accelerating: (1) a random forest tree for face detection task, and (2) convolution neural networks for image classification task.

4.1 High Level Synthesis

High Level synthesis (HLS), also known as C synthesis, behavioral synthesis or algorithmic synthesis, is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. Synthesis begins with a high-level specification of the problem, where behavior is generally decoupled from e.g. clock-level timing. The desired hardware's behaviour (algorithm) can be implemented in a high level language like C/C++ and then RTL/hardware is automatically generated using a vendor specific compiler.

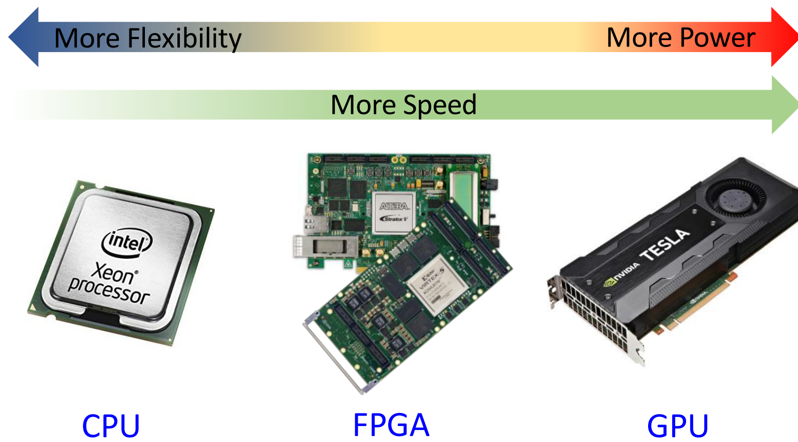


Figure 4.1: Comparison between CPU, FPGA and GPU for deep learning inference. CPUs are the most programmable and thus are flexible to support all possible algorithms. But the flexibility comes at the cost of poor performance. GPUs on the other hand have highly threaded architecture and thus have the highest performance. They are also programmable to support multiple algorithms. However, GPUs are very power hungry. FPGAs with their programmable fabric are used to create custom hardware for any algorithm and thus have performance much better than CPUs while their power consumption is much smaller compared to GPUs.

HLS tools are gaining importance among FPGA design community as they accelerate the design process. HLS lets hardware designers efficiently build and verify hardware, by giving them better control over optimization of their design architecture, and enabling the designer to describe the design at a higher level of abstraction while the tool does the RTL implementation. It can also bridge the gap between algorithm and hardware development. Moreover as the tool takes in C/C++ codes as inputs, functional validation is a lot faster compared to RTL validation. Fig. 4.2 demonstrates HLS design flow. HLS compilers are vendor specific and are optimized for specific FPGA boards. In this work, we look at two frameworks in particular, Altera OpenCL compiler and Xilinx HLS compiler.

4.1.1 *Altera OpenCL Framework*

In this section we will look at Altera's HLS framework. There is a recent interest in using OpenCL (Suda *et al.* (2016)), a C-based programming language, for FPGAs because of its parallel programming model. Moreover, the same OpenCL codes can easily be ported to different platforms: CPUs, GPUs, DSPs or heterogeneous systems consisting of a combination of them.

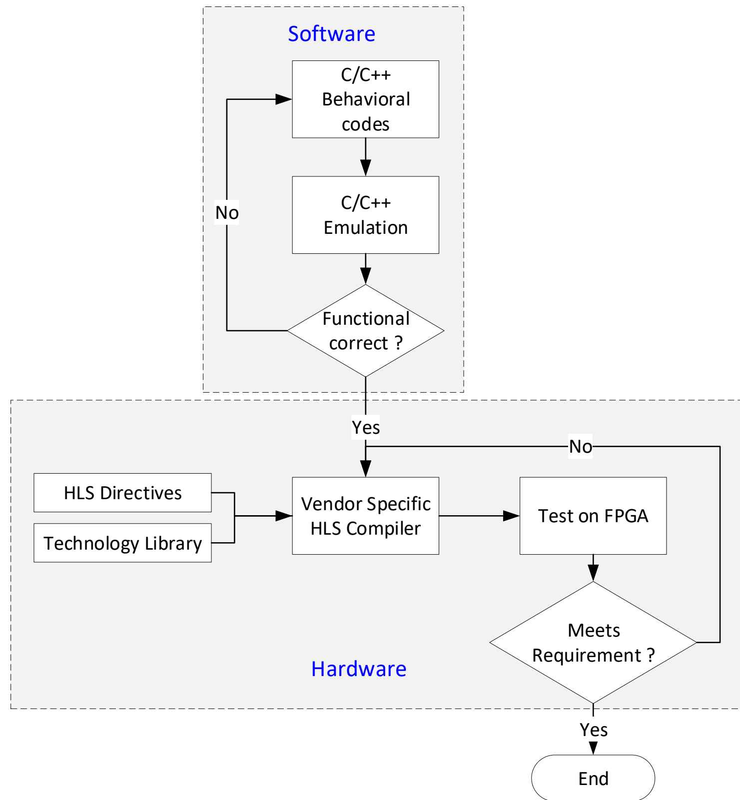


Figure 4.2: Design flow for high level synthesis (HLS). First the C++ codes are made functionally correct with fast emulation mode of HLS. After that HLS directives are used to explore architectural space till we satisfy the throughput and power requirements. C++ emulation and compiling high level codes to RTL ensures fast prototyping and low turnaround time to market.

OpenCL compilers not only compile an OpenCL code to RTL, but also integrate it with the interfacing IPs for external memory and for communication between host CPU and FPGA accelerator board. They abstract the designer/user from the intricacies of traditional FPGA design flow such as RTL coding, integration with interfacing IPs and timing closure, which considerably reduces the design time, while achieving performance comparable to the traditional flow, but possibly at the expense of higher on-chip memory utilization (Abdelfattah *et al.* (2014)).

The design flow of the OpenCL based FPGA accelerator used in this work is shown in Fig. 4.3. It consists of a FPGA accelerator board that is integrated into the PCIe slot of a desktop CPU that acts as the OpenCL host. In general, OpenCL framework consists of two components (a) an OpenCL code that is compiled and

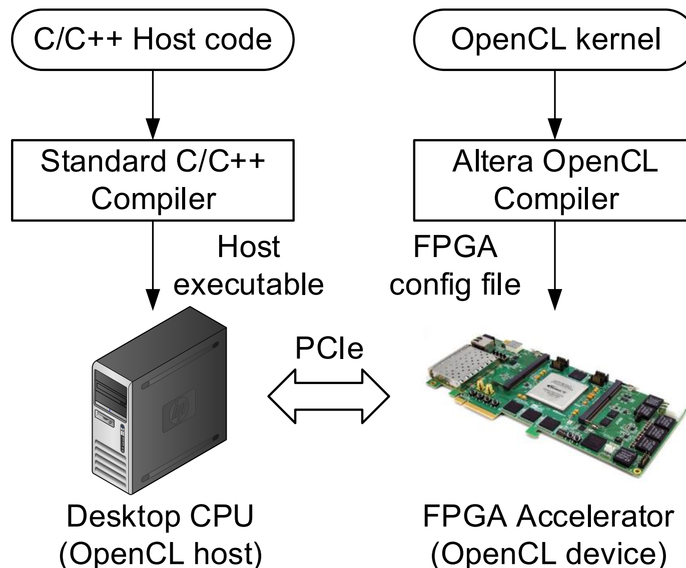


Figure 4.3: Design flow of OpenCL based FPGA accelerator. The heterogeneous system consists of two parts: (a) Host CPU running C/C++ codes, (b) FPGA accelerator device programmed with RTL file generated using openCL kernel. The host and the device communicate using PCIe port. The host executes the main task and offloads compute intensive portions of the application to FPGA accelerator.

synthesized to run on the FPGA accelerator and (b) a C/C++ based host code with vendor-specific application program interface (API) to communicate with the FPGA board. Since the host code is generic C/C++ codes, it can take advantage of libraries like openCV, blas etc for a lot of tasks. In this design flow, the host (CPU) is used for trivial tasks, as a controller for the FPGA accelerator and is responsible for data transfer to/from FPGA device. The compute intensive parts of the application are offloaded to the FPGA device to accelerate execution. The tool-kit provides support for emulation, which runs the OpenCL code on host CPU, thus allowing for quick functional verification before going to the full FPGA implementation.

The Altera SDK for OpenCL provides different synthesis constructs to enable acceleration of OpenCL kernels such as loop unroll factor and Single-Instruction-Multiple-Data (SIMD) vectorization factor. It also has constructs for choosing the number of hardware threads (work group size) working in parallel. All these constructs can be used as knobs for design space exploration to optimize for area, power or performance.

4.1.2 Xilinx HLS Framework

Xilinx's HLS platform (Vivado HLS) offers the designer more flexibility and access to lower level hardware details. It can be used to convert both OpenCL kernels and C/C++ codes to RTL. Creating specialized hardware from C/C++ is achieved by inserting specific compiler directives in the source code. The heterogeneous accelerator architecture in this case is similar to what we saw in the previous section. FPGA is still a slave device which is controlled by an general purpose CPU. In this case, both the FPGA and CPU sit on the same die and are connected by AXI Bus.

Xilinx HLS supports 3 types of AXI ports: (1) AXI Master, (2) AXI Slave and (3) AXI Stream. The AXI slave ports are generally used by the host for programming

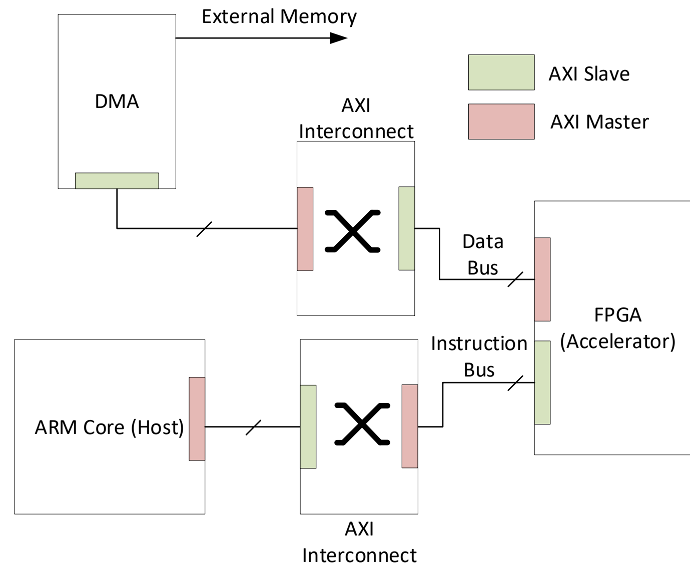


Figure 4.4: System diagram of C based HLS accelerator from Xilinx. The host (master) and the accelerator (slave) both sit on the same SoC using AXI bus. The master is responsible for transferring data to the accelerator and programming it. The FPGA accelerator has both AXI master (for high performace DMA transfers from external memory) and AXI slave interfaces (handshake protocol with CPU master).

the accelerator and for the handshake signals. The AXI master ports are used by the accelerator to initiate high performance DMA transfers to/from external memory. AXI Stream ports are useful when we have a constant source streaming in data to the accelerator. A typical system with the host and accelerator in this framework is shown in Fig. 4.4. Few of the key optimization directives used in Vivado HLS are presented in Table 4.1.

In the next sections, we shall discuss two example designs which use these HLS frameworks to accelerate two AI applications. First we shall look at a design which does high performance face detection with CPU-FPGA acceleration. Then we shall look into a design that implements a throughput-optimized OpenCL based FPGA accelerator for large scale convolution neural networks.

Table 4.1: Vivado HLS Key Optimization Directives. These directives are added to functionally correct C/C++ codes and then compiled to hardware. Keeping hardware in mind, these directives can be used to derive highly threaded hardware which can be optimized for performance, power and area. This method removes the need to code cycle-to-cycle detailed RTL and thus improves prototyping time drastically.

Directive	**Description**
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently.
DEPENDENCE	Provides additional information that can overcome loop-carry dependencies and allow loops to be pipelined.
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.

4.2 Face Detection using Random Forest Tree

Past few decades has witnessed evolutionary developments in Human-Computer-Interaction systems and computer vision. Face detection, which can be done effortlessly by humans, is considered a fundamental part of such systems. However, diversity in the nature of human faces (e.g., size, location, pose, orientation and expression) along with the changes in the environmental conditions (e.g., illumination, exposure, occlusion, etc.) makes face detection a challenging task. Detecting a large, unknown number of faces in a single frame of photograph, taken in realistic application scenarios, involves complex algorithms for segmentation, extraction and verification of possible facial features from an uncontrolled background. All these make face detection a very computationally expensive task to achieve real time performance with sufficient accuracy.

Being a mature and classic topic in the field of computer vision, many algorithms have been proposed to address this problem. Among the best performing are algorithms based on the classic Viola and Jones architecture (Viola and Jones (2001)) such as Deformable Parts Models, Headhunter model (Mathias *et al.* (2014)) and convolution neural network based algorithms (Li *et al.* (2015), Farfade *et al.* (2015)). We use the state-of-the-art [1] rigid templates based detector (Mathias *et al.* (2014)), which achieves highest precision and recall rates compared to other reported models, and accelerate it in a heterogeneous system (CPU+FPGA) to achieve real-time detection rates without significant degradation in accuracy.

[1] At the time this work was done Mathias *et al.* (2014) was state-of-the-art. Better performing algorithms have been proposed since then.

4.2.1 Algorithm background

The architecture of the face detection model is shown in Fig. 4.5, which consists of scaling, channel generation, integral channel computation, classifier output computation and non-maximum suppression stages. The classifier stage comprises of 5 trained models each with 2,000 weak classifiers and consumes significant time in the model evaluation. The baseline face detection model from in this acceleration framework, which uses 10 feature maps (channels) consisting of 6 quantized histogram of gradient (HOG) orientation channels, 1 magnitude gradient and 3 color channels (LUV color space) (Dollár *et al.* (2009)). The input features used by the detector are simple rectangular pooling regions. For fast computation of these rectangular features we use integral images proposed by Viola and Jones face detector (Viola and Jones (2001)). The model uses 2,000 shallow boosted weak classifiers with a depth of 2 in each of the 5 trained models: 1 frontal face model, 2 side views and 2 mirrored models. The classifier combines the outputs of the entire weak classifiers and compares it with a threshold to give the bounding box for a face along with a score. All the 5 trained models are evaluated separately for each window. The input image is scaled up and down with scaling factors ranging from 0.2× to 3× to enable detection of faces with broad range of sizes. The bounding boxes from all the scales are passed through a non-maximum suppression (NMS) stage, which keeps only one detected bounding box per face by selecting the highest score detection and removing the redundant overlapping boxes with lower scores.

4.2.2 FD Accelerator Design

Fig. 4.6(left) shows the major computational blocks involved in this algorithm. The computational time breakdown of each of these blocks for an input color image

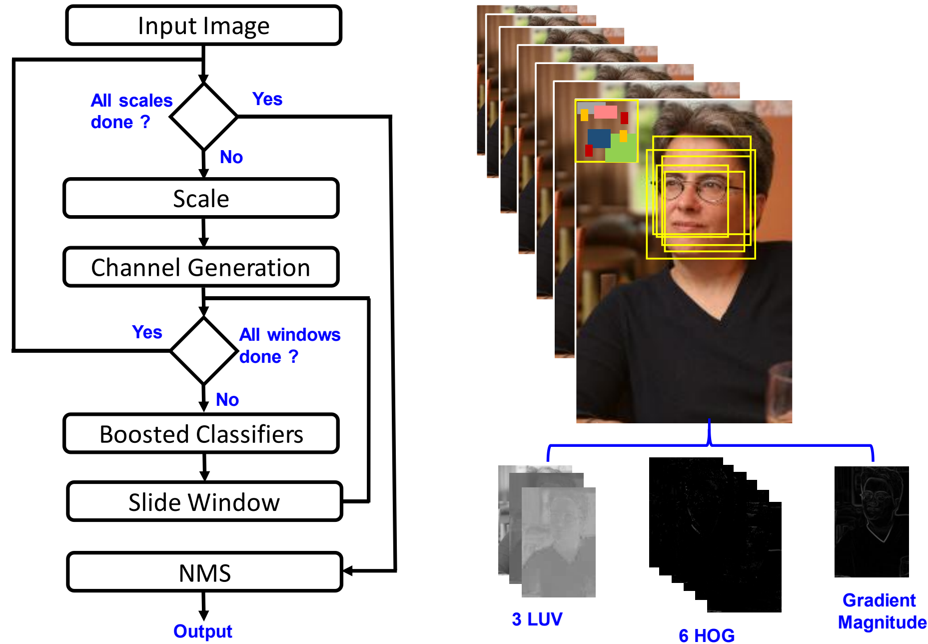


Figure 4.5: Architecture of face detection model in Mathias *et al.* (2014). They use 30 difference scales of the original image (for faces of different sizes) and 10000 week classifiers (for faces with different orientations). A total of 10 channels are generated using the scaled image. A scanning window with a rigid template based classifier is used to detect faces. Non Maximal suppression (NMS) is then used to remove redundant detections and preserve the best detections for final box drawing.

of dimension 320240 pixels on a general purpose CPU (Intel(R) Core i5-4590 @3.30 GHz with 32GB Memory) is shown in Fig. 4.6(right). Even though each weak classifier involves very simple operation, the overall iterative computation of 2,000 classifiers constitutes for 91.5% of the total time. This emphasizes the need for hardware acceleration of classifier computation. We mainly focuses on the FPGA acceleration of the classifier computation, whereas the remaining non-critical stages are computed in the host CPU.

Acceleration techniques in hardware implementation of the face detection algo-

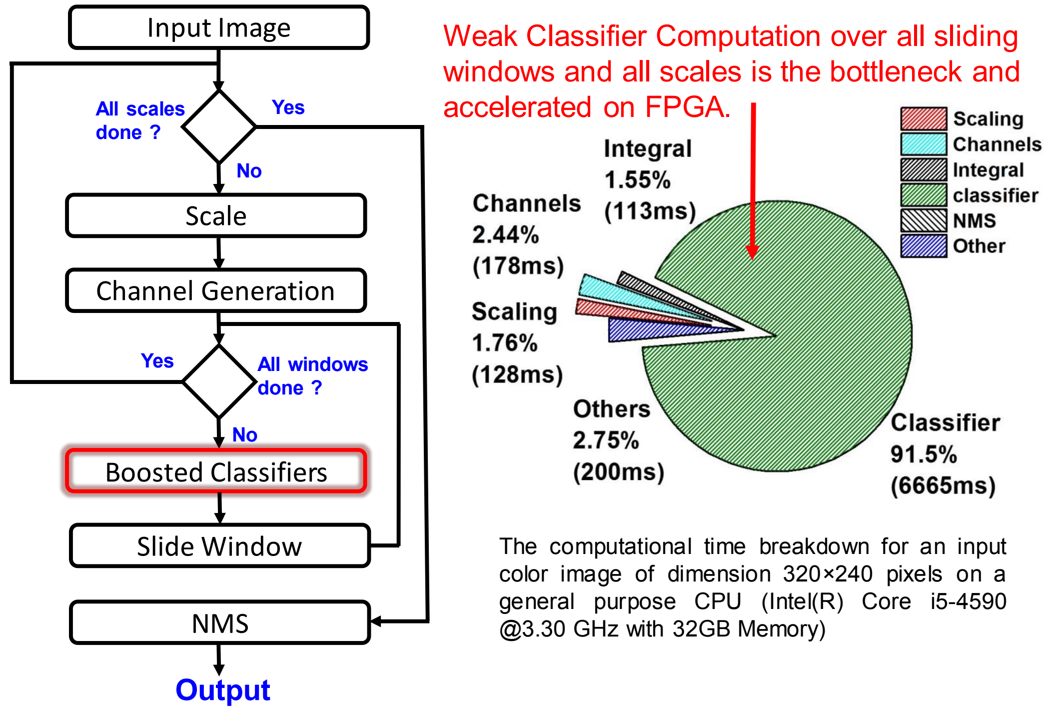


Figure 4.6: Time profiling of face detection algorithm on Intel Core i5-4590 CPU. Computation of 10,000 weak classifiers at all positions on the 30 different scaled versions of the input image is the most time consuming part. It consumes $\sim 91\%$ of the total time. So the heterogeneous system was designed so that the FPGA device will accelerate the boosted classifier computation while the CPU handles the rest.

rithm can be broadly classified into two categories: (a) Acceleration by coarse computation and (b) Hardware acceleration by parallel computation of outputs. The headhunter baseline model from Mathias *et al.* (2014) used here uses multiple input scaling factors ranging from 0.2 to 3.0 with a step of 0.1. It also defines a sliding window stride of 1 at all scales. Such fine grain scaling factor step size and sliding window strides result in large number of computations and hence increase time for model evaluation without necessarily increasing the detection quality. In this work, we use resized AFW database to evaluate the optimal scaling factor step size and slid-

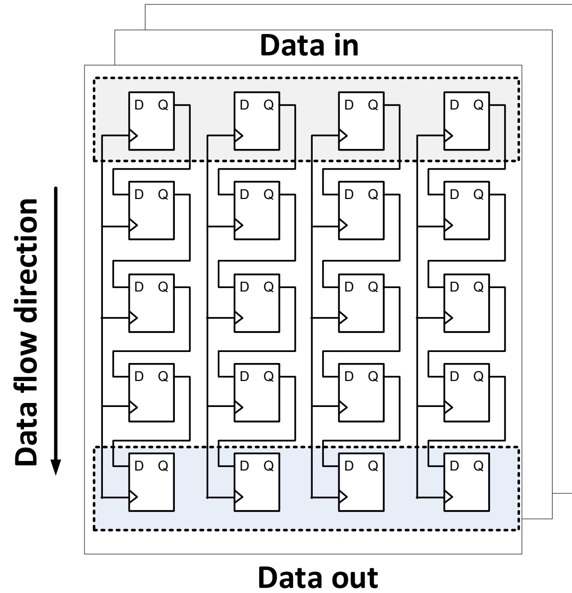


Figure 4.7: Shift register implementation to store the channel data. This architecture allows loading only the new row while the rest of window is reused from previous iteration, thus saving 95% of the integral data transfer time.

ing window stride without significant reduction in the detection quality. At the larger scaled versions of the input image, which are aimed at detecting small faces, majority of the windows have background data. Thus the useful information is sparsely available in those scales. In comparison to these, the smaller scales compress the useful information into a small area. So using a fixed stride does not give the best performance when a trade-off can be made between accuracy and speed. An adaptive stride technique is implemented in this work which keeps the stride to 1 at lower scales and gradually increases at larger scales.

From the hardware perspective, the operations can be accelerated by using multiple parallel classifier cores with separate local memory banks. This enables the cores and sub cores to operate independently on different parts of the same problem. However, since each classifier core uses the integral data from all the 10 input features,

the integral data has to be replicated as many times as the number of parallel classifiers. Hence the amount of on-chip memory available on the FPGA board defines the maximum number of parallel classifier cores in such a nested architecture. On the other hand, as shown in Fig. 4.7, using shift register based logic to read the integral channel data helps reduce the data transfer time by loading only the new row of data while reusing rest of the window data from the previous iteration. For example, for a window size of 20×20, shift register based sliding window implementation illustrated in Fig. 4.7 allows loading only the new row (1×20) while the rest of window (19×20) is reused from previous iteration, thus saving 95% of the integral data transfer time.

Rigid template based detectors use pooling of rectangular areas as input features. In the ones using integral channels, these features are computed by simple operations of 2 additions and subtractions. Each time a feature is computed, 4 points from one of the 10 channels are used and the output of this is compared with a threshold, θ_1, to decide which of the remaining 2 nodes should be computed next. The computation in the 2nd node, which is similar to 1st node except for different data points, is compared with a threshold, θ_2, and the output from the weak classifier is determined. Weights from all 2,000 weak classifiers are summed and compared with a threshold, θ_f, to give a strong classification result. These operations are performed on for every window for the 5 templates and 18 scales of the input image. For a QVGA (320×240) image, there are 96,714 windows upon which 10,000 classifiers are to be computed. The pseudo-code for this algorithm is shown in Algorithm 1. Thus, even though the computations involved are simple, the throughput of the system is bottle necked by the memory access bandwidth.

One way to mitigate this bottleneck is by having multiple copies of the same data and compute features in parallel. However, the features require random data points from a given window thereby forcing us to make copies of the entire window

Algorithm 1 Pseudo-code for face detection algorithm.

```
 1: procedure DETECT_FACE(input_image, weights)
 2:     Apply smoothing filter and downscale image by 4.
 3:     for each scale do
 4:         Compute the 10 channels
 5:         Integrate the 10 channels
 6:         for each sliding window do
 7:             for each template do
 8:                 for each weak classifier do
 9:                     Compute stage 1 = a_1 + d_1 - b_1 - c_1
10:                     Compare with Stage 1 threshold, θ_1
11:                     Determine Stage 2 node
12:                     Compute stage 2 = a_2 + d_2 - b_2 - c_2
13:                     Compare with Stage 2 threshold, θ_2
14:                     Update running sum of weight
15:             Compare with threshold θ_f and determine if it is a face
16:     Do NMS on detections from all scales to remove redundancies
17:     return Detections
```

data. The limited amount of on-chip SRAM in FPGAs poses a limit on the number of such parallel units. In our experimental results (Table I), we show that as we increase the number of parallel classifiers performance increases, which is highest with 4 parallel classifiers. After that, the increase in the number of parallel units degrades performance till the point where we cannot fit the design into the FPGA board. Due to memory limitations in FPGAs, all of the image data except the current operating window is stored in the external DDR memory. Since the access to DDR has very high latency and power consumption, most of the optimization techniques were aimed at reducing the number of DDR access transactions and reusing the loaded data for multiple windows before discarding it.

4.2.3 Results

In this section, we present the implementation results (Mohanty *et al.* (2016)) of the headhunter baseline face detection model on Altera Stratix-V A7 FPGA based DE-5 board using Altera OpenCL software development kit (SDK). The Stratix-V A7 FPGA consists of 622K logic elements, 256 DSP blocks and 2,560 M20K RAMS, whereas there are $2 \times 2GB$ DDR3 DRAMs present on the board that function as the global memory.

Fig. 4.8 presents the performance of the system (left) and resource utilization of the FPGA (right) for different number of parallel computing classifier units. A performance improvement of 40% is observed when the classifiers are computed by two

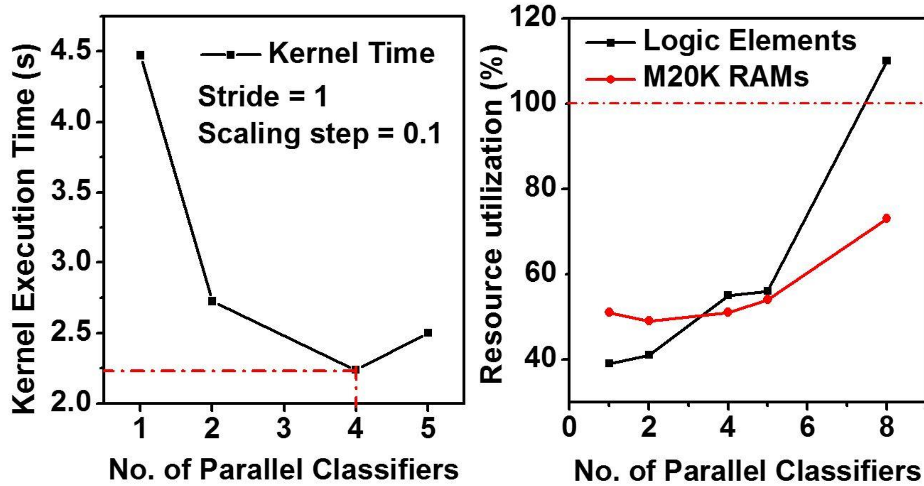


Figure 4.8: Performance (left) and FPGA resource utilization (right) for different number of parallel compute classifier units. Execution time reduces with increase in number of parallel classifiers till 4, after that it increases because of memory access contention between the parallel execution units. From the resource utilization plot we see that 4 parallel classifiers can be accommodated in the given FPGA device with resource utilization around 50%.

units in parallel. Increasing the number of parallel units, increases performance till the number of classifiers is 4 after which it degrades. This is because limited memory bandwidth leads to contention between the computing units sharing the memory that results in pipeline stalls. For a given FPGA, the maximum parallelization achievable is limited by the logic elements and M20K RAMs.

Table 4.2 shows the total time and classifier computation time on the FPGA using the proposed acceleration techniques described previously. As can be seen, increasing the stride of the sliding window provides substantial improvements. In low scaling factors, however, the image has very dense information, thus sliding the window with a stride of two leads to poor accuracy. On the other hand, the adaptive stride scheme gives much better performance. Here, we kept stride in x direction equal to 1 for all the lower scales and increased it successively to a value of 4 for higher scales. The stride in y-direction is fixed to 2. As seen, we were able to reduce the classifier computation time to 145ms. The precision-recall curves of the FPGA implementation tested on AFW database scaled down to 320×240 pixels for different

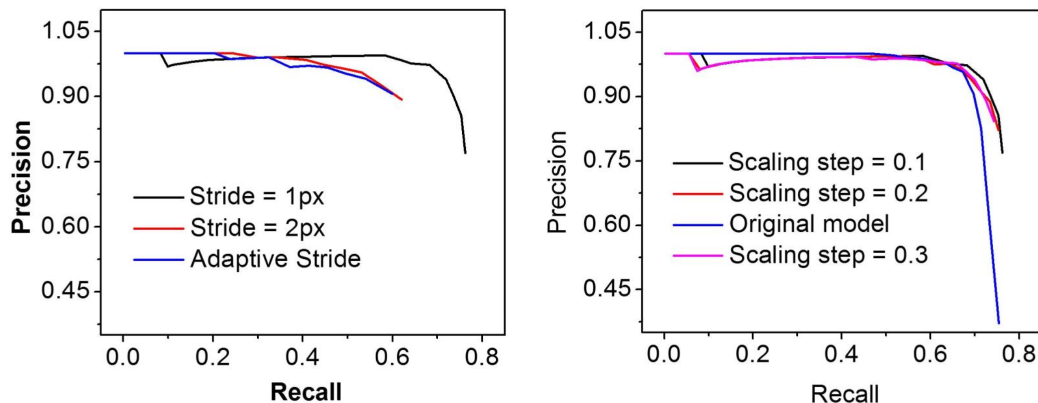


Figure 4.9: Precision vs recall curves of the CPU+FPGA implementation of the model tested on downsized AFW database using (a) different strides and (b) different scaling factor step sizes.

Table 4.2: Face detection run time comparison. 4 parallel classifiers operating in parallel with adaptive stride scheme and step size of 0.2, achieves $\sim 30\times$ improvement in speed over CPU with minimal reduction in detection accuracy.

Implementation	$N_{\mathbf{CLASS}}$	**Stride**	**Step Size**	**Total Time (s)**	**Kernel Time (ms)**
CPU Only	-	1	0.1	7.284	-
CPU + FPGA	1	1	0.1	4.472	4279.11
CPU + FPGA	1	1	0.2	2.049	4962.28
CPU + FPGA	2	1	0.1	2.726	2542.32
CPU + FPGA	2	1	0.2	1.256	1165.71
CPU + FPGA	4	1	0.1	2.237	2044.07
CPU + FPGA	4	1	0.2	1.029	937.522
CPU + FPGA	4	2	0.1	0.73623	547.946
CPU + FPGA	4	2	0.2	0.34838	251.354
CPU + FPGA	4	Adaptive	0.1	0.48803	300.284
CPU + FPGA	4	Adaptive	0.2	0.23780	145.001

strides and different scaling factor step sizes are shown in Fig. 7. The baseline has lower precision and recall than that reported in Mathias *et al.* (2014), because of the use of the down-scaled AFW database. From Fig. 4.9 and Table 4.2, we conclude that using a scaling step size of 0.2 along with adaptive stride that increases with the scaling step size gives the best performance without significant degradation in precision and recall.

Thus in heterogeneous platforms consisting of CPU and FPGA performance-critical classifier stage can be implemented on FPGA, whereas the non-critical stages cna be evaluated on the host CPU. Classifier acceleration is achieved by exploring a combination of multiple acceleration techniques such as coarse computation by using larger sliding window stride and scaling factor step size, performing multiple classifiers in parallel and integral data reuse by shift register based implementation. These techniques achieve a speed up of 30x compared to a CPU implementation, without significant degradation in precision or recall.

4.3 Convolution Neural Networks

Convolutional Neural Networks (CNNs), inspired by visual cortex of the brain, are a category of feed-forward artificial neural networks. As discussed previously, CNNs are primarily employed in computer vision applications such as character recognition (LeCun *et al.* (1990)), image classification (Krizhevsky *et al.* (2012), Szegedy *et al.* (2015), Simonyan and Zisserman (2014)), video classification (Karpathy *et al.* (2014)), face detection (Li *et al.* (2015)), gesture recognition (Barros *et al.* (2014)), etc., are also being used in a wide range of fields including speech recognition (Abdel-Hamid *et al.* (2014)), natural language processing (Collobert and Weston (2008)) and text classification (Lai *et al.* (2015)).

The operations in CNNs are computationally intensive with over billion operations

Table 4.3: Operations in AlexNet CNN Model (Krizhevsky *et al.* (2012)). To classify a 200×200 input image ~ 1.9GOPs are required. Convolution and fully connected layers dominate in terms of operation requirement.

Layer	**#Features**	**Input Dimension**	**Kernel Size**	**#Operations**
Input Image	3	224	–	–
Convolution-1/ReLU-1	96	55	11	211120800
Normalization-1	96	55	–	3194400
Pooling-1	96	27	3	629856
Convolution-2/ReLU-2	256	27	5	448084224
Normalization-2	256	27	–	2052864
Pooling-2	256	13	3	389376
Convolution-3/ReLU-2	384	13	3	299105664
Convolution-4/ReLU-2	384	13	–	224345472
Convolution-5/ReLU-2	256	13	–	149563648
Pooling-5	256	6	3	82944
Fully Connected-6/ReLU-6	4096	–	–	75501568
Fully Connected-7/ReLU-7	4096	–	–	33558528
Fully Connected-8	1000	–	–	8192000
Total Operations				1455821344

per input image (Szegedy *et al.* (2015)), thus requiring high performance server CPUs and GPUs to train the models. As can be observed in Table 4.3, total number of operations needed to classify a 220×220 image using AlexNet (Krizhevsky *et al.* (2012)) is humongous. However, they are not energy efficient and hence various hardware accelerators have been proposed based on FPGA. FPGA based hardware accelerators have gained momentum owing to their reconfigurability and fast development time, especially with the availability of high-level synthesis (HLS) tools from FPGA vendors. Moreover, FPGAs provide flexibility to implement the CNNs with limited data precision which reduces the memory footprint and bandwidth requirements, resulting in a better energy efficiency (e.g. GOPS/Watt).

4.3.1 CNN Accelerator Design

In this section, we will look into the implementation of a CNN accelerator using OpenCL HLS framework for FPGA. In particular, we shall look at the critical design variables that impact the throughput and are used for optimization. This is achieved by analytically modelling of various CNN layers are functions of these design variables. Then we shall take a look at how thoughput can be systematically improved subject to constraints like FPGA logic utilization, on-chip memory and external memory bandwidth. The results are demonstrated with two CNNs, AlexNet (Krizhevsky *et al.* (2012)) and VGG (Simonyan and Zisserman (2014)).

In our FPGA design, we first developed computing primitives of CNNs using OpenCL framework. A scalable convolution module is designed based on matrix multiplication operation in OpenCL, so that it can be reused for all convolution layers with different input and output dimensions. Similarly, we developed scalable hardware modules for normalization, pooling, and fully-connected layers. We identified key design variables such as loop-unroll factor and SIMD vectorization factor,

which determine hardware parallelization and thus directly impact the throughput, external memory bandwidth requirement, and computational resource utilization.

Intuitively, assigning more computational resources to performance-critical operations in convolution and fully connected layers would maximize the overall throughput of the system. However, it may not be a global optimal solution, because each layer has different feature dimensions and the computational resources are limited. Hence, there is a great need for a design space exploration methodology that maximizes the throughput by optimally distributing the FPGA resources among various scalable CNN hardware blocks.

We propose a design space exploration framework based on both analytical and empirical models of CNN layer performance and resource utilization, to find the optimal values of the key design variables that maximize the throughput of a generic CNN model on a given FPGA board with limited computation resources, on-chip memory, and external memory bandwidth.

3-D Convolution as Matrix Multiplication

Convolutions are the most performance-critical operations in CNNs, involving computationally intensive 3-D multiply and accumulate (MAC) operations of the input features with the convolution weights as given in Equation 2.1. To maximize the overall throughput of the accelerator and also make the design portable to any other CNN model, a scalable convolution block is needed such that the data can be iterated through it in software.

We implemented the scalable convolution block by mapping the 3-D convolutions as matrix multiplication operations similar to that in Chellapilla *et al.* (2006) by flattening and rearranging the input features. As an example, Fig. 4.10 illustrates how Convolution-1 layer in AlexNet is mapped from 3 input features with dimensions

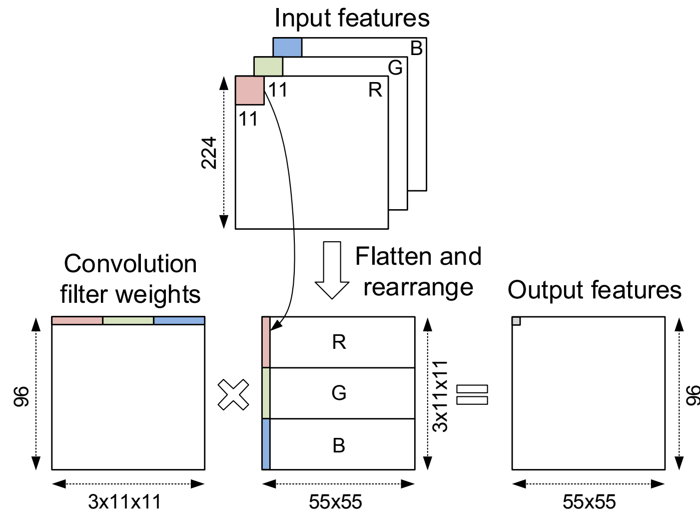


Figure 4.10: Mapping 3D convolutions to matrix multiplications.

224×224 to a rearranged matrix with dimensions of $(3 \times 11 \times 11) \times (55 \times 55)$. The input features from the first convolution window of 11×11 are flattened and arranged vertically as shown in Fig. 4.10. Similarly, the entire rearranged matrix can be generated by sliding the 1111 convolution filter across the input features. After input features are rearranged, the convolution operation transforms to a generic matrix multiplication operation. Note that we perform the input feature rearrangement on-the-fly by storing them in the FPGA on-chip memory before performing matrix multiplication, which reduces the external memory requirement by eliminating the

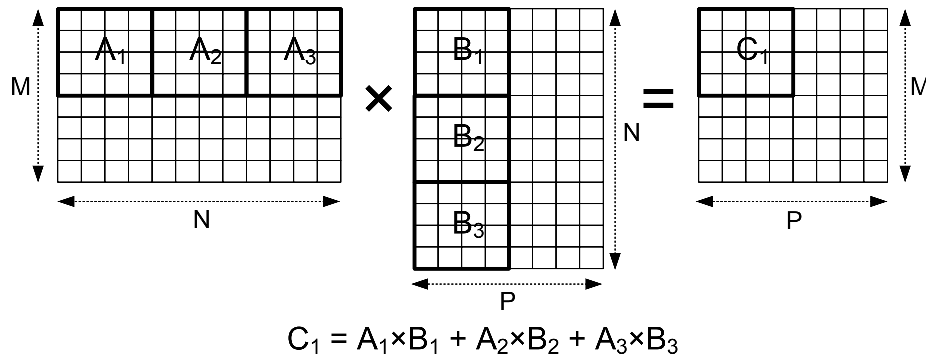


Figure 4.11: Accelerating matrix multiplications in OpenCL.

need to store the entire rearranged input feature matrix.

The pseudo-code for matrix multiplication based convolution implementation in OpenCL is shown in Algorithm 2. It can be summarized as the following three basic operations which are repeated over each row of the weight matrix.

1. Fetch the convolution weights to the local memory.
2. Compute input feature actual address locations and fetch them to local memory.
3. Compute N_{CONV} multiply and accumulate operations in parallel.

We utilized matrix multiplication OpenCL code from Altera's OpenCL matrix multipication tutorial and appended the input feature rearranging operation. Understanding the matrix multiplication OpenCL implementation is critical for acceleration of the convolution operation. The implementation of matrix multiplication operation in OpenCL is illustrated in Fig. 4.11, which consists of convolution weight matrix A ($M \times N$), multiplied by the rearranged input feature matrix B ($N \times P$) to compute the

Algorithm 2 Pseudo-code for convolution implementation.

1: **procedure** CONVOLUTION($input_feature_map, weights$)
2: Get current work-item/thread identifiers (x, y).
3: **for** $each\ N_{CONV}\ elements\ width-wise\ in\ weight\ matrix$ **do**
4: Compute address locations for input features and weights
5: Fetch input features to inputs[x][y] in local memory
6: Fetch convolution weights to weights[y][x] in local memory
7: Wait till $N_{CONV} \times N_{CONV}$ inputs and weights are loaded
8: **for** $all\ x\ and\ y\ (compute\ N_{CONV}\ MAC\ operations\ in\ parallel)$ **do**
9: $convolution\ output = convolution\ output + weight[x][k] \times input[y][k]$
10: Wait till all work-items complete computation on fetched data
11: Save convolution output to output buffer.

output feature matrix C ($M \times P$). It consists of $N_{CONV} \times N_{CONV}$ threads or OpenCL work-items, which fetch the first $N_{CONV} \times N_{CONV}$ inputs to the local memory where $N_{CONV} = 4$ in this example. Each work-item performs N_{CONV} parallel multiply and accumulate (MAC) operations on the local memory data, which is accomplished by loop unrolling that replicates the hardware resources for acceleration. This process is repeated by sliding the $N_{CONV} \times N_{CONV}$ window column-wise in matrix A and row-wise in matrix B and performing the MAC operations to get $N_{CONV} \times N_{CONV}$ elements in the product matrix C.

From Fig. 4.11, we see that the input and output matrix dimensions must be a multiple of N_{CONV}, which might not always be possible because of different number of input and output features and different feature dimensions in different convolution layers. Hence we use zero padding in the input matrices to make their dimensions a multiple of N_{CONV}. Increasing N_{CONV} boosts the throughput as it fetches larger number of inputs to the local memory and performs computations on them without having to wait for external data. On the other hand, it increases the logic utilization and execution time if the zero-padding is excessive in some layers. We use SIMD vectorization factor (S_{CONV}), as another design variable to accelerate the convolution operation, which represents the factor by which computational resources are vectorized to execute in a Single-Instruction-Multiple-Data fashion. This factor improves the throughput by a factor of S_{CONV}. Depending on the model configuration parameters such as number of features and their dimensions and the number of CNN layers, choosing an appropriate combination of (N_{CONV}, S_{CONV}) maximizes the overall throughput of the CNN.

Normalization Layer

Local response normalization (LRN) implementation requires an exponent operation as shown in Equation 2.2, which is expensive to precisely implement in hardware. Hence we implement the exponent function $f_1(x_o)$ shown in Equation 4.1 using a piece-wise linear approximation function $pwlf(x_o)$.

$$out(f_o, x, y) = in(f_o, x, y).f_1(x_o) \tag{4.1}$$

$$f_1(x_o) = (1 + x_o)^{-\beta}; x_o = \frac{\alpha}{K} \sum_{f_i = f_o - K/2}^{f_o + K/2} in^2(f_i, x, y) \tag{4.2}$$

Here K represents the number of features used for normalization. Using the AlexNet model data as an example, the exponent function $f_1(x_o)$ is approximated using a piece-wise linear function using 20 points with a maximum error of 1%. Because of the wide dynamic range of values involved in x_i computation, normalization is implemented in 32-bit floating point representation. The exponent function and the piece-wise linear approximate function along with the approximation error are plotted

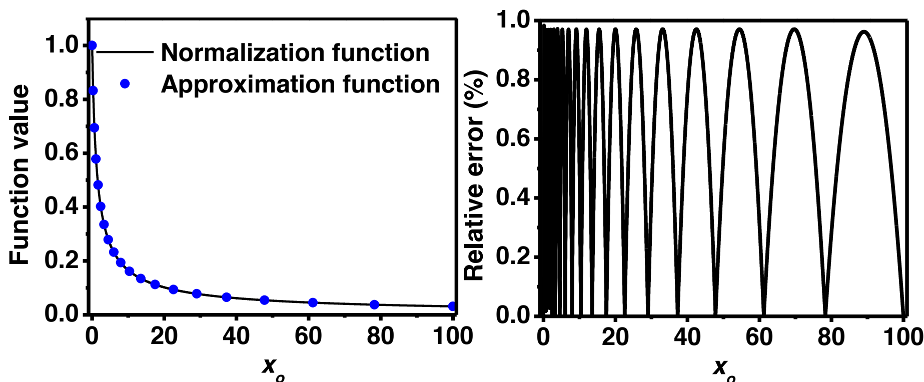


Figure 4.12: Piece-wise linear approximation of normalization operation kernel with a maximum error of 1%. Using piece-wise lookup tables, normalization is performed without the need for expensive hardware for performing non-linear functions.

in Fig. 4.12. Normalization is implemented as a singlethreaded OpenCL code using loop unroll factor (N_{NORM}), which represents the number of normalization operations it performs in a single cycle. The Altera OpenCL compiler automatically infers pipelining whenever there are no data dependencies between multiple iterations. It uses local memory to store the sum of squares of a sliding window of K input features, while performing the normalization operation on the computed sum of squares using the piece-wise linear approximation function, $pwlf(x_o)$.

Implementation of other Layers

Pooling is implemented using a single work-item kernel where acceleration is achieved by unrolling the loop to generate N_{POOL} parallel outputs in a single cycle. Fully-connected layer or inner-product layer is also implemented as single work-item kernel, where acceleration is achieved by performing NFC parallel multiply and accumulate operations, which accelerates the performance by a factor of NFC. Nonlinear activation function ReLU, which performs the function $y = max(x, 0)$ is incorporated at the output of convolution and inner product implementations with a flag to enable or disable it.

4.3.2 Design Space Exploration

Choosing the best combination of the design variables (N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}) that maximizes the performance of the CNN accelerator, while still being able to fit in the limited FPGA resources is a non-trivial task, which emphasizes the need for a systematic design space exploration methodology. Optimization framework that relies on full FPGA synthesis at each design point may not be feasible especially because of the long run time, which could take hours, or potential synthesis failures that occur due to utilization of hardware resources. Hence we model the

performance and resource utilization and use them for fast design space exploration.

In this section, we first formulate the optimization problem and present the analytical and empirical modeling of the performance and FPGA resource utilization as a function of the design variables for each CNN layer.

Problem Formulation

The resource-constrained throughput optimization problem can be formulated as follows.

$$Minimize \sum_{i=0}^{TL} runtime_i(N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}) \tag{4.3}$$

$$Subject\ to\ \sum_{j=0}^{L} DSP_j \leq DSP_{MAX} \tag{4.4}$$

$$\sum_{j=0}^{L} Memory_j \leq Memory_{MAX} \tag{4.5}$$

$$\sum_{j=0}^{L} Logic_j \leq Logic_{MAX} \tag{4.6}$$

where TL represents the total number of CNN layers including the repeated layers, L denotes the total number of CNN layer types and $runtime - i$ is the execution time of the layer-i. DSP_{MAX}, $Memory_{MAX}$, and $Logic_{MAX}$ represent the total DSP, on-chip memory and FPGA logic resources, respectively, available in a given FPGA.

Performance Modeling

The execution time of each CNN layer is analytically modeled as a function of the design variables and validated by performing full synthesis at selective design points and running them on the FPGA accelerator. The execution time of convolution layer-i is modeled as follows.

$$Convolution\ Runtime_i = \frac{No.\ of\ Convolution\ Ops_i}{N_{CONV} \times S_{CONV} \times Frequency_i} \tag{4.7}$$

where PAD_{NCONV} ceils its inputs to the multiple of N_{CONV}. Maximum frequency of the kernel, which is also a function of N_{CONV} and S_{CONV}, is modeled empirically from the synthesis data with different random seeds, as shown in Fig. 4.13. The execution time model and the measured execution time of convolution layers 1-5 of AlexNet implementation for a sweep of N_{CONV} at different S_{CONV} values are compared in Fig. 4.14.

Other layers

Similarly, the execution time of normalization, pooling and fully connected layers are modeled as functions of their respective loop unroll factors used for acceleration as follows.

$$Runtime_i = \frac{\#Operations_i}{Unroll\ factor \times Frequency} \tag{4.8}$$

The execution time model vs. measured run time of normalization and fully connected classification layers are shown in Fig. 4.15.

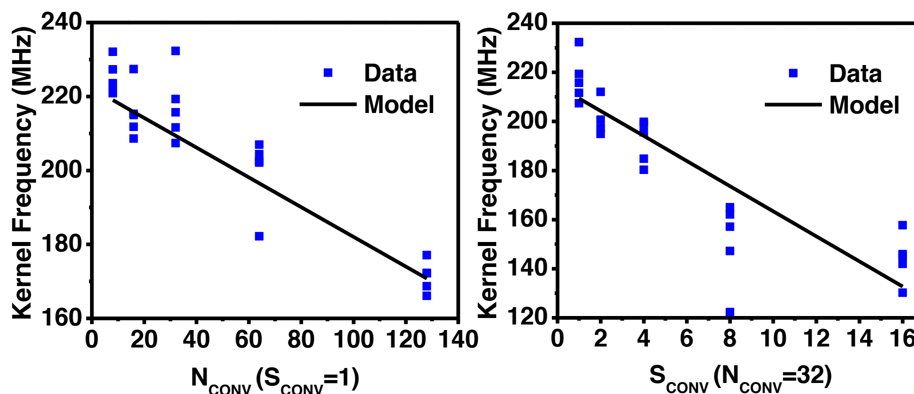


Figure 4.13: Kernel frequency modeling from full synthesis data at 5 random seeds. RMS error of the fit: 12.57 MHz

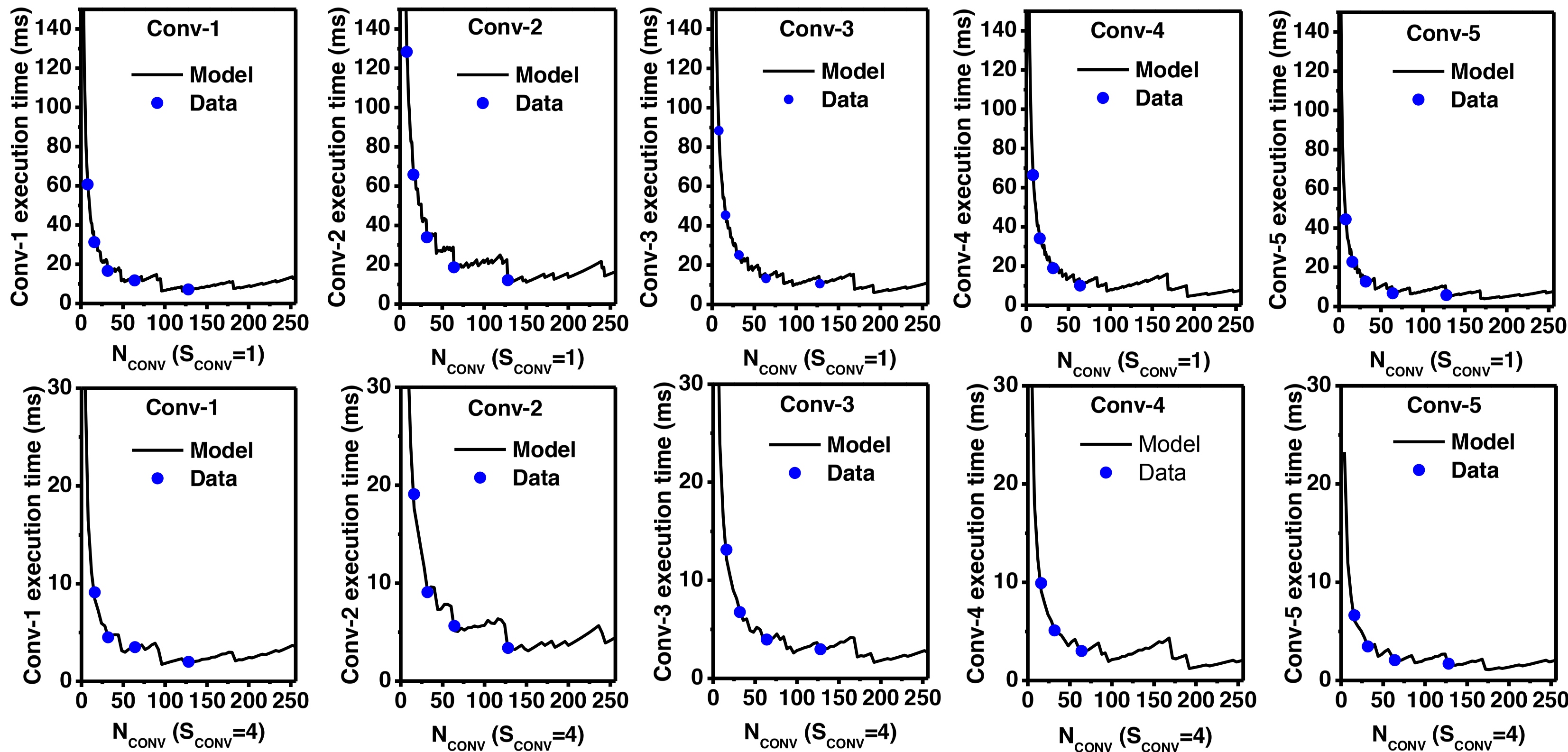


Figure 4.14: Run time model vs. measured time of convolution layers 1-5 for a sweep of matrix multiplication block size (N_{CONV}) for SIMD vectorization factor, $S_{CONV} = 1$ and 4.

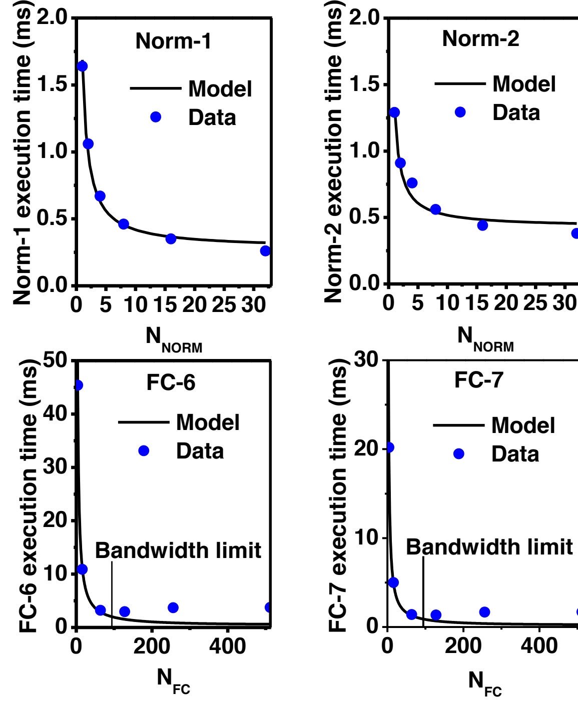


Figure 4.15: The execution time model vs. measured data of normalization and fully connected layers in AlexNet for sweep of loop unroll factors N_{NORM} and N_{FC}.

Memory Bandwidth

Input data, weights, intermediate data and final output data are stored in the external memory that is present on the FPGA accelerator board. To enable efficient data transfer to and from external memory, Altera OpenCL compiler generates complex load/store units similar to those in GPUs, which combine multiple external memory accesses into a single burst access, known as memory coalescing. This ensures the efficient use of available external memory bandwidth with less contention for memory accesses between multiple computational blocks. On the other hand, this makes it difficult to model the external memory bandwidth usage with respect to the design variables used for acceleration. This problem is aggravated by the reuse of the scalable hardware blocks in multiple iterations of CNN layers with different input dimensions,

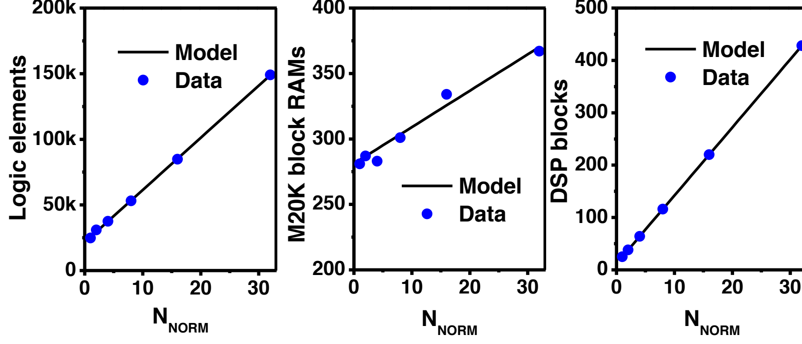


Figure 4.16: Resource utilization empirical models for normalization block.

which will have different access patterns. For example, the execution time of fully connected layers 6 and 7 of AlexNet model shown in Fig. 4.15 shows that the model matches well with the measured time till $N_{FC} = 100$. For NFC¿100, the measured time increases slightly, but the model still shows a reduction in execution time. This discrepancy is caused by the bandwidth limitation of the FPGA board used for the model validation. Hence we use the bandwidth limitation of the FPGA board to define the upper limits for the design variables in our optimization framework.

Resource Utilization Modeling

Analytically modeling the FPGA resource utilization of an algorithm in a high-level language such as OpenCL may not be feasible because of the optimizations performed in the HLS tools. Hence, we use synthesis results to empirically model the FPGA resource utilization. DSP block usage, on-chip memory and logic utilization from synthesis results of each CNN layer are fitted to linear regression models as a function of their design variables.

For example, resource utilization models of normalization block are shown in Figure 12. Logic element and DSP utilization from the synthesis data in Figure 12 show a linear increase with the swept design variable N_{NORM}. On the other hand, on-chip

memory utilization model shows small discrepancy with the synthesis data at intermediate points because of implementation of coalescing load/store units in which the memory resource utilization depends on whether the external memory data width is an integer multiple of the design variables i.e. N_{NORM}.

4.3.3 Optimization Framework

From the convolution run time model in Fig 4.14, we see that it is non-monotonic, because of the differences in dimensions of the CNN layers. Although exhaustive search of all the design variables could be done using the performance and resource utilization models, it may not be feasible if the number of design variables and/or the FPGA resources increase substantially. This calls for global optimization methodologies such as simulated annealing, genetic algorithm or particle swarm optimization with integer variables and multiple inequality constraints. In this work, we use genetic algorithm with integer constraints from the global optimization toolbox in Matlab for the design space exploration. Genetic algorithm is a stochastic optimization technique that mimics the biological evolution process and is popularly used to find the global minimum of an objective function subject to a set of constraints. It can also handle mixed integer programming problems, where some of the design variables are integers. It iteratively improves the quality of the solution by generating a set of candidate solutions at each iteration or generation from a combination of the best solutions from the previous generation based on a set of genetic rules selection, crossover and mutation. The solutions that violate the constraints (i.e. Equations 4.3 - 4.6) resources are penalized in such a way to ensure convergence of the feasible solutions to a global minimum.

The design space of the OpenCL-based FPGA accelerator design is illustrated below.

$$S_{CONV} = 1, 2, 4, 8 or 16 \tag{4.9}$$

$$N_{CONV} = N \times S_{CONV}, 0 < N < N_{MAX} \tag{4.10}$$

$$0 < N_{NORM} < N_{NORM(MAX)} \tag{4.11}$$

$$0 < N_{POOL} < N_{POOL(MAX)} \tag{4.12}$$

$$0 < N_{FC} < N_{FC(MAX)} \tag{4.13}$$

where all the design variables are integers, and upper limits of the design space exploration such as N_{MAX}, $N_{NORM(MAX)}$, $N_{POOL(MAX)}$, and $N_{FC(MAX)}$ are determined by the external memory bandwidth of the FPGA board. For example, in a fully connected layer implementation where k bytes are required for each MAC operation, N_{FC} of an accelerator board with external memory bandwidth of M_{BW} is computed as shown in Equation 4.14.

$$F_{FC(MAX)} = \frac{Memory\ bandwidth\ (M_{BW})}{k \times Frequency} \tag{4.14}$$

For an FPGA system with 6 GB/s external memory bandwidth, requiring 2 bytes per MAC operation in a fully connected layer with 100MHz kernel frequency, the upper limit for NFC can be computed from Equation 4.14 as 30. Similarly, the upper limits of other blocks can be computed based on the number of external memory transfers required for each operation.

4.3.4 Results

In this section, we present the validation results of the proposed optimization framework by implementing and accelerating two large-scale CNN models: AlexNet and VGG-16 models on two FPGA boards with different hardware resources. The

Table 4.4: Comparison of FPGA accelerator boards.

Specification	**P395-D8**	**DE5-Net**
FPGA	Stratix-V GSD8	Stratix-V GXA7
Logic elements	695k	622k
DSP blocks	1,963	256
M20K RAMs	2,567	2,560
External memory	$4 \times\ 8GB\ DDR3$	$2 \times 2GB\ DDR3$

hardware specifications of the two Altera Stratix-V based boards are summarized in Table 4.4.

Both networks are implemented in OpenCL with fixed-point operations using 8-bit weights for convolution and fully connected layers as obtained from the precision study in Chapter 3. Although 10-bit precision is chosen for inner product weights, they are still represented using 8-bits as the 2 bits in MSB side are zeros in all the weights. Using the performance and resource utilization models and the maximum hardware resources available in the two boards, optimization framework is run on both AlexNet and VGG models to find the optimal combination of design variables (N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}) that maximizes the throughput. For example, Fig. 4.17 shows the execution time of the best solution of each iteration during the optimization of AlexNet implementation on DE5-Net FPGA board. Table 4.5 shows the execution time from the model, measured execution time on FPGA and the FPGA resource utilization at chosen points A, B and C in Fig. 4.17. The final design variables for both networks optimized for the two FPGA boards are shown in Table 4.6. VGG model does not include normalization layers, hence the corresponding kernel is removed for the FPGA implementation.

Using Altera OpenCL SDK, the OpenCL kernel codes for AlexNet and VGG

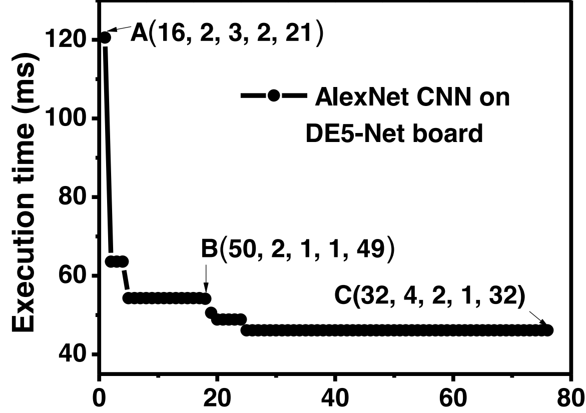


Figure 4.17: Optimization progress of AlexNet implementation. Design variables (N_{CONV}, S_{CONV}, N_{NORM}, N_{POOL}, N_{FC}) are shown at points A, B and C.

models are compiled for the two boards using the corresponding optimized parameters from Table 3. Using the host code APIs, FPGA is programmed and the CNN model is run by queueing the OpenCL implemented CNN kernels with appropriate arguments that consist of input/output buffer address locations and the layer dimensions. The execution time of each kernel and the entire model are measured and throughput is computed as (total number of operations)/(execution time).

The total classification time per image and overall throughput of AlexNet and VGG models on P395-D8 and DE5-Net boards are compared with Caffe tool (Jia *et al.*

Table 4.5: Summary of Execution time and Utilization.

	A	**B**	**C**
Exec. time (model)	120.6 ms	54.3 ms	46.1 ms
Exec. time (measured)	117.7 ms	52.6 ms	45.7 ms
Logic elements	158k	152k	153k
M20K memory blocks	1,439	1,744	1,673
DSP blocks	164	234	246

Table 4.6: Optimized parameters.

	P395-D8 board		**DE5-Net**	
	AlexNet	**VGG**	**AlexNet**	**VGG**
N_{CONV}	64	64	32	64
S_{CONV}	8	8	4	2
N_{NORM}	2	-	2	-
N_{POOL}	1	1	1	1
N_{FC}	71	64	32	30

Table 4.7: Classification time/image and overall throughput.

	FPGA	**Classification time/image (ms)**	**Throughput (GOPS)**
AlexNet	**P395-D8**	20.1	72.4
	DE5-Net	45.7	31.8
	CPU	191.9	7.6
VGG	**P395-D8**	262.9	117.8
	DE5-Net	651.2	47.5
	CPU	1437.2	21.5

(2014)) running on Intel core i5-4590 CPU (3.3 GHz) as shown in Table 4.7. Although both FPGAs have similar number of logic elements and on-chip memory blocks, the smaller number of DSP blocks in DE5-Net accounts for its lower throughput compared to that of P395-D8. The software implementation in Caffe tool uses libraries optimized for basic vector and matrix operations (i.e., ATLAS Whaley and Dongarra (1998)) for performing CNN operations. Our OpenCL based FPGA implementations on P395-D8 achieve 9.5× and 5.5× speedups for AlexNet and VGG models, respectively, compared to the CPU implementation in Caffe tool.

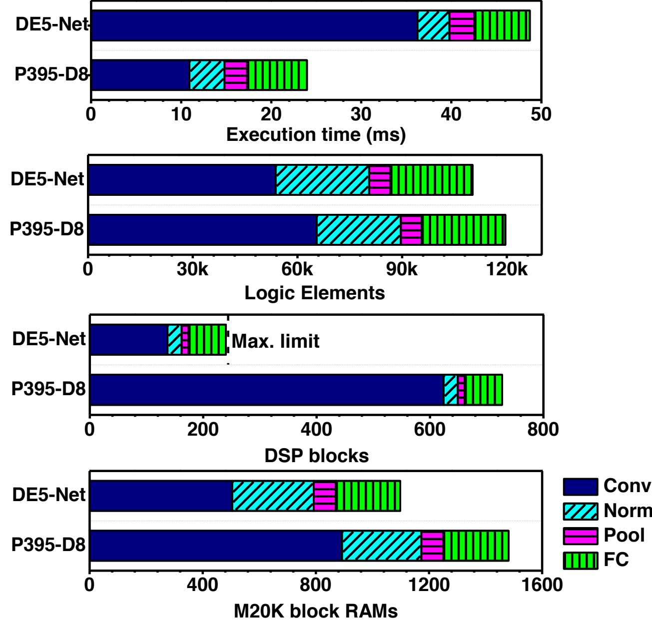


Figure 4.18: Execution time and resource utilization of each CNN layer type for AlexNet implementation on P395-D8 and DE5-Net FPGA boards.

The execution time, throughput and the resource utilization of each kernel type of the AlexNet implementation on P395-D8 and DE5-Net FPGA accelerator boards are shown in Fig. 4.18. VGG implementation on P395-D8 achieves a peak throughput of 136.5 GOPS for convolution layers, and 117.8 GOPS including all layers and operations while performing image classification. From the implementation results, we see that throughput of the accelerator is largely proportional to the number of DSP blocks used in the implementation. AlexNet implementation on P395-D8 board is limited by the number of available M20 block RAMs, while only 727 out of 1963 available DSP blocks are utilized. On the other hand, throughput on DE5-Net FPGA board is limited by the lower number of available DSP blocks, although the on-chip memory resources and logic elements are not fully utilized.

Our optimization framework reports the hardware resource that causes the performance bottleneck, such that the user can choose another FPGA hardware, which

Table 4.8: Model accuracy comparison.

Accuracy	Full precision in Caffe tool		Fixed-point FPGA implementation	
	Top-1	**Top-5**	**Top-1**	**Top-5**
AlexNet	56.82%	79.95%	55.41%	78.98%
VGG	68.35%	88.44%	66.58%	87.48%

has larger number of the specific hardware resources (e.g. DSP blocks). This methodology can also be used to find the ideal specifications of an FPGA suited for CNN, by performing optimization with relaxed constraints for the bottleneck hardware resource. For example, increasing the onchip memory resources on P395-D8 FPGA by 10% directly increases the throughput of AlexNet implementation by 10%. This work assumes that MAC operations are implemented using the DSP blocks only. However, we can potentially enhance the throughput further by using the remaining logic elements to implement MAC operations, which will be studied in future work.

The top-1 and top-5 accuracies of FPGA implementation of AlexNet and VGG models compared to those of the full-precision Caffe models are summarized in Table 4.8. The accuracy degradation due to fixed-point operations in FPGA implementation is ¡2% for top-1 accuracy and ¡1% for top-5 accuracy for both AlexNet and VGG models. Both DE5-Net and P395-D8 boards are connected to a PCIe slot of a desktop computer whose CPU operates as the OpenCL host. Since the FPGA board receives power from external power port as well as PCIe slot, the power measurement of the FPGA

Both DE5-Net and P395-D8 boards are connected to a PCIe slot of a desktop computer whose CPU operates as the OpenCL host. Since the FPGA board receives power from external power port as well as PCIe slot, the power measurement of the FPGA board itself is not straightforward. We attempted to block the power

connection through PCIe and have the FPGA board powered only through the external power port. This way, the average power consumption of DE5-Net board was measured as 24.2W after programming AlexNet configuration, and as 25.8W while performing classification. On the other hand, the same measurement method was not feasible on P395-D8 board as it was designed to use both power supplies. Nonetheless, we measured its power consumption as 19.1W after programming with AlexNet configuration file, using a utility function provided by board manufacturer that measures the steady state power of the board.1 We compare the performance of VGG model implementation on P395-D8 FPGA board to the existing FPGA based CNN accelerators in Table 7. For the entire VGG model with 30.9 GOP, our FPGA accelerator achieves overall throughput of 117.8 GOPS for ImageNet classification.

4.3.5 Conclusion

In this work, we implemented scalable CNN layers on FPGA using OpenCL framework and identified the key design variables for hardware acceleration. Further, we proposed a design space exploration methodology based on a combination of analytical and empirical models for performance and resource utilization, to find the optimal design variables that yield maximum acceleration of any CNN model implementation using limited FPGA resources. Using the proposed methodology, we implemented two large-scale CNNs, AlexNet and VGG, on P395-D8 and DE5-Net FPGA boards and achieved superior performance compared to previous work.

Chapter 5

HARDWARE SOFTWARE CO-OPTIMIZATION

Deep neural networks have demonstrated significant performance improvements in many AI applications. While this accuracy improvement is ground breaking, it comes at huge computational costs. For example, going from resNet-34 to resNet-152 decreases top-1 error from 21.84% to 19.87% (mere 1.97% reduction) at the expense of increase in operations from $\sim$ 8 G-Ops to $\sim$ 23 G-Ops (15G-Ops / $\sim$ 300% increase in computations). Fig. 5.1 illustrates the operation requirements and the accuracy numbers of some of the popular and top performing neural networks. As observed, for the algorithm developers the ultimate goal has been to obtain the highest accuracy in a multi-class classification problem framework, regardless of the actual inference time. Since there is no incentive in speeding up inference time, practical applications of these models are affected by resource utilization, power-consumption, and latency.

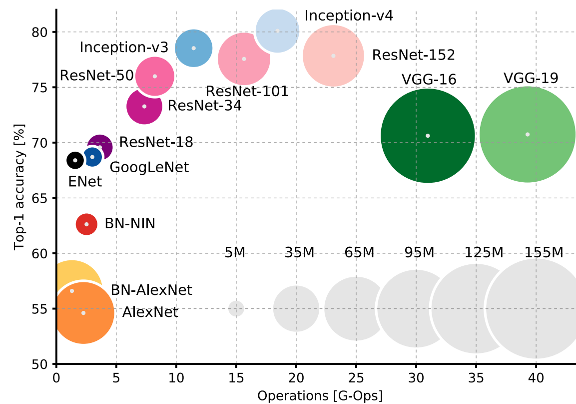


Figure 5.1: Top1 vs. operations, size $\propto$ parameters (Canziani *et al.* (2016)). Newer network architectures are much more efficient with respect to model size and the number of operations required.

Hardware designers on the other hand have treated the algorithm as black box and focused on optimizing the hardware. Relentless efforts have resulted in rapid progress in the field of deep learning hardware. Several high performance custom hardware accelerators have been developed to efficiently execute AI algorithms Farabet *et al.* (2009), Farabet *et al.* (2011), Chen *et al.* (2015c), Kadetotad *et al.* (2015), Seo *et al.* (2015), Kadetotad *et al.* (2014), Xu *et al.* (2014), Mohanty *et al.* (2017), Kim *et al.* (2017). However, these hardware architectures were conceptualized, developed and optimized keeping in mind the software algorithms that are supposed to be executed on them. While break-throughs in hardware performance has been achieved, there is still a big room for improvements given the algorithms are developed keeping hardware in mind. In this chapter we shall look at some similar examples where the gap between hardware and software can be bridged leading to huge improvements in hardware performance while minimal effects on algorithm's accuracy.

Rest of the chapter is organized as follows. In section 5.1 we will look at an design example where the algorithm is modified to a form best suited for hardware acceleration. We consider *Non-Maximal Suppression* algorithm (NMS) as a case study to demonstrate hardware software co-optimization. NMS is widely used in many computer vision algorithms to remove redundancies. Its also used in *proposal layer* for faster RCNN network for object detection. We propose a novel hardware friendly NMS algorithm which removes the need for sorting in hardware and reduces computation complexity from $O(nlog(n))$ to $O(n)$.

5.1 Non-Maximal Suppression

Non-maximal suppression in object detection neural networks in the task of finding all non-overlapping proposals with $score > theshold$, where score is the probability that the detection is an object and the threshold is minimum probabilty value for any

Algorithm 3 Pseudo-code for Non-Maximal Suppression algorithm.

1: **procedure** NMS($box_coordinates, scores, thresh$)
2: Get all box co-ordinates and scores.
3: Calculate all areas $\rightarrow (x_2 - x_1 + 1) \times (y_2 - y_1 + 1)$
4: Sort all boxes according to scores $\rightarrow order = scores.argsort()[:: -1]$
5: Create an empty list keep $\rightarrow keep = [\ ]$
6: **while** order has elements **do**
7: Get the first index $\rightarrow i = order[0]$
8: Put the corresponding box in keep array $\rightarrow keep.append(i)$
9: Get overlap $x_1 \rightarrow xx_1 = maximum(x_1[i], x_1[order[1:]])$
10: Get overlap $y_1 \rightarrow yy_1 = maximum(y_1[i], y_1[order[1:]])$
11: Get overlap $x_2 \rightarrow xx_2 = maximum(x_2[i], x_2[order[1:]])$
12: Get overlap $y_2 \rightarrow yy_1 = maximum(y_2[i], y_2[order[1:]])$
13: Get overlap height $\rightarrow w = np.maximum(0.0, xx_2 - xx_1 + 1)$
14: Get overlap width $\rightarrow h = np.maximum(0.0, yy_2 - yy_1 + 1)$
15: Get overlap area $\rightarrow inter = w \times h$
16: Get IoU overlap $\rightarrow ovr = inter/(areas[i] + areas[order[1:]] - inter)$
17: Get proposals with less overlap $\rightarrow inds = np.where(ovr <= thresh)[0]$
18: Keep proposals with less overlap $\rightarrow order = order[inds + 1]$
19: **return** $keep$

detection be considered as containing an object. The pseudo code with corresponding python code for NMS shown is given in Algorithm 3. As observed, NMS procedure can be broadly divided into two major tasks:

1. **Sorting**: Sort all the proposals/boxes according to their scores,

2. **Suppression**: Starting from the top scored box, keep removing boxes with lower scores and high overlap with higher scored box

5.1.1 NMS computation complexity

In this section, we shall take a deep dive into the computation complexity of software baseline NMS algorithm provided in Algorithm 3. In computer vision applications, NMS is generally applied to reduce the huge number of all possible boxes to a few highly probable true positive [1] boxes. This process is critical because because it reduces the number of times we need to perform the computation expensive fully connected layers to classify the box into one of the many classes. NMS is also used in post-processing to select the top scoring and non-overlapping boxes in the final proposals. When NMS is used inside a layer in the neural network (e.g. proposal layer in Faster-RCNNGirshick (2015)) the dimension of inputs to NMS can be huge ($\sim 1\ million$). This can potentially make NMS an computation bottleneck if its not carefully optimized.

Phase 1 of NMS implements sorting. Assuming that the software implements quick sort or merge sort, the time complexity of sorting is given by:

$$Time\ Complexity\ Sorting = O(nlog(n)) \tag{5.1}$$

Phase 2 of NMS is the suppression phase. In this phase, a top scoring box is

[1] A box is considered a true positive if the network labels it as an object and there is an actual object in the box

compared with every other lower scored box in the list for IoU overlap. Since in the worst case of suppression phase, every box can be compared with every other box, the time complexity of suppression phase is given by:

$$Time\ Complexity\ Suppression = O(m^2) \tag{5.2}$$

where, m is size of the list containing boxes for suppression phase.

NMS is generally associated with parameters called pre-nms-top-N and post-nms-top-N. Pre-nms-top-N defines how many top scored boxes after sorting are considered for suppresion (phase 2). Since suppression phase has quadratic complexity (eq. 5.2) pre-nms-top-N is generally fixed at a smaller number ($\sim 5K - 10K$). The acutal value is generally fixed empirically so as to minimize computation while not affecting final accuracy numbers.

Post-nms-top-N defines how many boxes are needed after NMS. This is also a critical parameter because it determines the computation time of both NMS and subsequent fully connected layers. More over, this parameter determines the maximum number of individual objects the network can detect in a given image. Like pre-nms-top-N, post-nms-top-N values are also empirically fixed to bring a balance between final accuracy and computation time. Typically its fixed at 300.

For a given algorithm with $pre-nms-top-N = k$ and $post-nms-top-N = m$, eq. 5.2 becomes:

$$Time\ Complexity\ Suppression = O(m \times k) \tag{5.3}$$

From eq. 5.1 and eq. 5.3, worst time complexity of NMS is given by:

$$Time\ Complexity\ NMS = O(nlog(n)) + O(m \times k) \tag{5.4}$$

Considering a design example with $n = 129360$, $m = 2000$ and $k = 300$, we get $nlog(n) \sim 661262$ and $m \times k \sim 600000$. Thus, in a typical design the time complexity

of NMS is determined by m,n and k. But for larger values of m and k, NMS execution time is proportional to $m \times k$.

5.1.2 Fast and Hardware Efficient NMS

In section 5.1.1, we saw that NMS has two sub-procedures performing sorting and suppression. Accelerating sorting in hardware is a non-trivial task. While CPUs are well suited for inherently sequential tasks like sorting, multi-threaded architectures like GPU and neural network accelerators are not good at it. Especially, when the input data is big, it has to be streamed into the core and is available one by one. This makes the sorting task even more in-efficient as when data cannot be cached, only viable sorting option is bubble sort which has a worst case complexity of $O(n^2)$. Also, because NMS is sometimes used in layers in the neural network, using the CPU for this purpose is no longer an option as that would involve interrupting the CPU within a single frame and transferring data from accelerator to CPU memory (which are very slow and inefficient). All these necessitates a more hardware friendly and efficient NMS algorithm.

From algorithm 3, after sorting we pick the top pre-nms-top-N proposals for suppression. This process of selecting the n top scored proposals after sorting reduces the number of proposals drastically, by selecting top $\sim$ 1.5% of all proposals. So the entire purpose of sorting in this procedure is to select the top 1.5% proposals. So if we consider the score associated with the last detection selected after sorting (algorithm 3) as a threshold value, the first phase of NMS algorithm can be thought of selecting all the proposals with scores above this threshold and using them for suppression phase. Lets call this as selection phase. To summarize, the new NMS algorithm should be able to do the following:

1. **Selection Phase**: Efficiently select the proposals with scores within top $\sim$ 1.5%

scores from a huge set of streaming proposals.

2. **Suppression Phase**: Efficiently suppress redundant overlapping proposals within the output from selection phase.

Selection Phase

So the task at hand, is to select proposals with scores within top $\sim 1.5\%$. Scores here are proportional to probability assigned by the neural network to the proposal for it encompassing a ground truth object. Since more than 95% of all proposals correspond to back ground, we can safely expect most of the scores (background related) to be

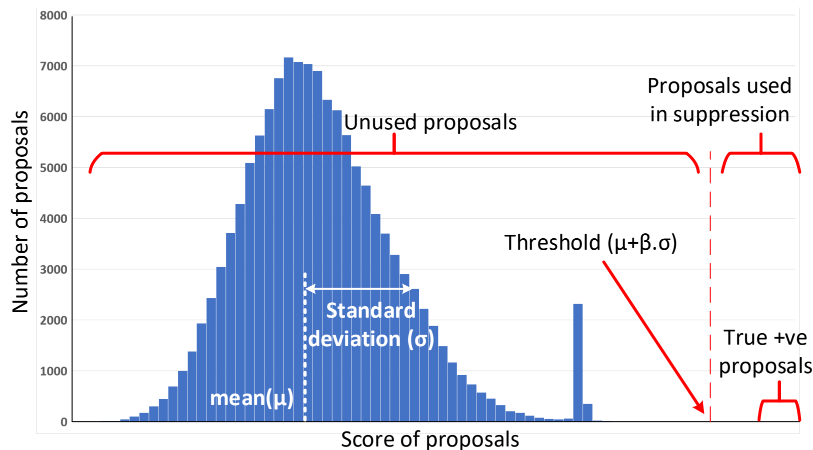


Figure 5.2: Distribution of proposal scores for a typical image in Faster-RCNN network. As very few anchors overlap properly with the ground truth objects and majority of the anchors have partial or no overlap, the distribution of the scores is gaussian with mean (peak) at low/negative scores. Top proposals which are necessary for correct detections are in the right tail region of the distribution. These proposals can be easily extracted by estimating mean(μ) and standard deviation(σ) of all scores and then discarding proposals with score smaller than $\mu + \beta \times \sigma$. β is a empirical parameter determined to minimize training dataset.

centered around a large -ve number and a very few scores (associated with ground truth objects) around some large +ve number. In between them its expected to be a smooth and continuous distribution of scores. So the distribution of scores can be considered as sum of two Gaussian: (1) one with a large peak, centered around negative scores, (2) one with a small peak, centered around positive scores. Since the Gaussian corresponding to proposals with true +ve proposals is typically very small compared to true -ve proposals, the sum of the two Gaussian above can be safely assumed to the same as the Gaussian corresponding to true -ve proposals with the true positive proposals lying in the right side tail of the Gaussian. This is illustrated using an typical design example in fig. 5.2.

Top scores can be extracted from the set by exploiting the property of the scores that they have a Gaussian distribution. Using the mean (μ) and standard deviation (σ) for the distribution, the top $\sim 1.5\%$ of proposals can be easily extracted by keeping only the proposals with scores above a threshold given by $\mu + \beta\sigma$. Here β is empirically estimated based on the percentage of top scored proposals needed in suppression phase. In general, β values in range $2.5 \sim 3$ yields good results.

Pseudo code for this is shown in Algorithm 4. To make NMS hardware efficient, we first stream in all the scores (for all proposals) and the threshold using mean and standard deviation. This process can be pipelined to process new data every clock cycle and has worst case complexity of $O(N)$. After that we stream all the proposals again. This time we use the threshold calculated before to ignore/throwaway low scored proposals. This effectively does what sorting accomplishes. But the proposals now are not in sorted order, so we cannot use the suppression algorithm from NMS as is to get the final result. Section 5.1.2 discusses the modified suppression algorithm to augment the proposal selection method described here.

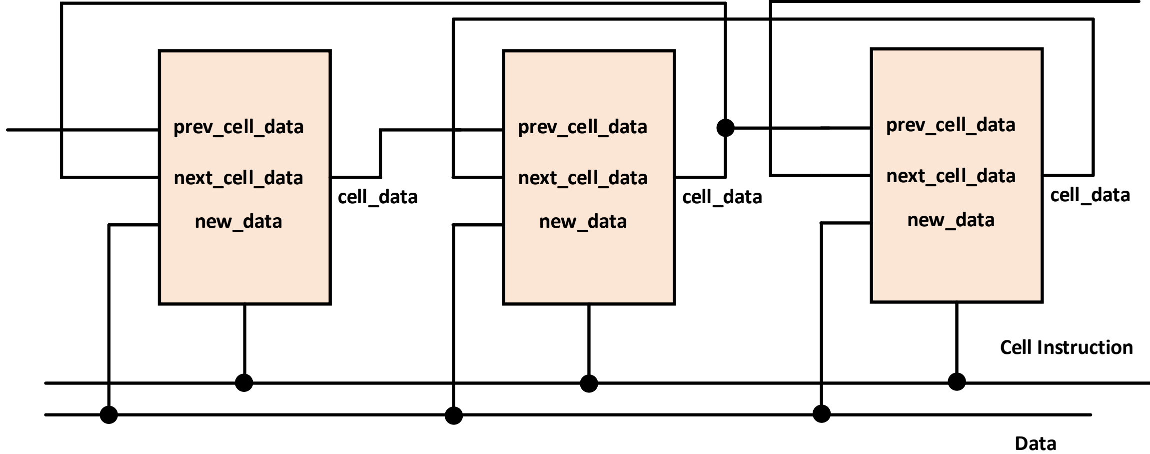


Figure 5.3: Proposed structure of internal register array for suppression phase of NMS. The cells are connected in a chained fashion to facilitate efficient ($O(1)$) insertion and deletion of new data. The cells store previous/next/current/new data based on the instruction that is provided to it using the instruction port.

Suppression Phase

Selection algorithm proposed in this work produces proposals with no specific order (un-sorted). So suppression phase has to do sorting so as to maintain order in the final result. For this, we propose to have a hardware register array to store the top selected proposals sorted according to their scores. Each cell is connected to two adjacent cells, (1) previous cell, with higher score and (2) next cell, with lower score. This chained connection of register cells is shown in fig. 5.3. As can be observed, each cell in the register array can receive data from 3 sources, (1) previous cell data, (2) the cell data itself and (3) next cell data. Each cell is capable of performing the following operations:

1. **Insert new data**: When a cell is inserting new data, the next cell receives data stored in this cell by registering data from *previous cell data* port. By multi casting this instruction to all cells downstream, $O(1)$ list insertion is achieved.

2. **Hold current data**: Holding data can be achieved by registering the current data stored in the cell. This does not change data stored in the register array.

3. **Delete current data**: When a cell is inserting new data, the cell receives data stored in next cell by registering the data from *next cell data* port. By multi casting this instruction to all cells downstream, $O(1)$ list deletion is achieved.

Pseudo-code for the proposed algorithm is given in Algorithm 4. The time complexity of the modified NMS algorithm, for m proposals after selection and n final proposals, is given by:

$$Time\ Complexity\ NMS = O(Selection) + O(Sorted\ Suppression) \tag{5.5}$$

$$Time\ Complexity\ Sorted\ Suppression = O(m \times n) \times O(insertion/deletetion) \tag{5.6}$$

With the proposed hardware register array, we can do insertion and deletion to the list in constant time. Worst case time complexity for sorted suppression can be obtained as:

$$Time\ Complexity\ Sorted\ Suppression = O(m \times n) \tag{5.7}$$

From 5.5 and 5.7, with the proposed changes in algorithm and hardware, we can very efficiently execute NMS on custom hardware accelerators with worst case complexity of $O(N) + O(m \times n)$. This has same complexity as the software baseline with sorting (5.4). Since custom hardware are can be optimized to the lowest level for the application achieving minimal wastage of clock cycles, the proposed hardware can be orders of magnitude higher in performance ($100\times \sim 1000\times$). Moreover, if the algorithm had not been modified to a variation more suitable to hardware architecture, we would have done sorting which has a complexity of $O(n^2)$, resulting in a very complicated hardware with very low performance ($\sim N\times$ slower, where

Algorithm 4 Pseudo-code for hardware efficient Non-Maximal Suppression.

```
1: procedure NMS(box_coordinates, scores, nms_threshold, min_area, num_std_dev(β))
2:     for prop_new in all proposals do
3:         Calculate sum of all scores → sum = Σ_{i=0}^{N} score_i
4:         Calculate sum of all squared scores → squared_sum = Σ_{i=0}^{N} score_i^2
5:         Get mean of all scores → μ = sum/N
6:         Get standard deviation of all scores → σ = sqrt(squared_sum/N − μ^2)
7:     Calculate score threshold → θ = μ + β × σ
8:     Instantiate a hardware list → keep = [ ]
9:     for prop_new in all proposals do
10:        if prop_new.score < θ then
11:            Ignore prop_new → GOTO step 9
12:        else
13:            Initialize a counter variable → j = 0
14:            for prop_keep in keep do
15:                if prop_new.score ≥ prop_keep[j].score then
16:                    Insert prop_new in keep list → keep.insert(prop_new, j)
17:                    Increment counter → j++
18:                    break → GOTO step 27
19:                else
20:                    Get IoU overlap between prop_keep[j] and prop_new
21:                    if IoU ≥ nms_threshold or prop_new.area ≤ min_area then
22:                        Ignore prop_new → GOTO step 9
23:                        Increment counter → j++
24:            if j < max_keep_size then
25:                Insert prop_new in keep list → keep.insert(prop_new, j)
26:                Get new proposal → GOTO step 9
27:            if j ≤ keep.size then
28:                while j ≤ keep.size do
29:                    Get IoU overlap between prop_keep[j] and prop_new
30:                    if IoU ≥ nms_threshold then
31:                        Delete proposal from keep list → keep.delete(j)
32:    return keep
```

N is the number of proposals). For complicated neural networks with large N, the proposal layer could alone take a few seconds, where as the optimized algorithm and hardware can be executed in a few milliseconds.

5.2 Conclusion

In this chapter, we demonstrated that, efforts to accelerate a given algorithm with hardware architecture optimizations doesn't always lead to the most efficient design. Even though the performance can be involved to a great extent with hardware optimizations, there is almost always a huge headroom for performance improvement that can be achieved by hardware-software co-optimization. To demonstrate this we used a very common algorithm used in deep neural network based object detection algorithms, non-maximal suppression, and identified the major bottlenecks that inhibit efficient hardware acceleration. We showed that mapping the algorithm as is to hardware will have worst case time complexity of $O(n^2)$ while the software complexity is $O(nlog(n))$. We replaced the bottleneck parts of the algorithm with procedures more suitable for hardware acceleration and showed that with the proposed method and hardware architecture, the same functionality can be achieved with time complexity of $O(n)$.

Chapter 6

BEYOND CMOS

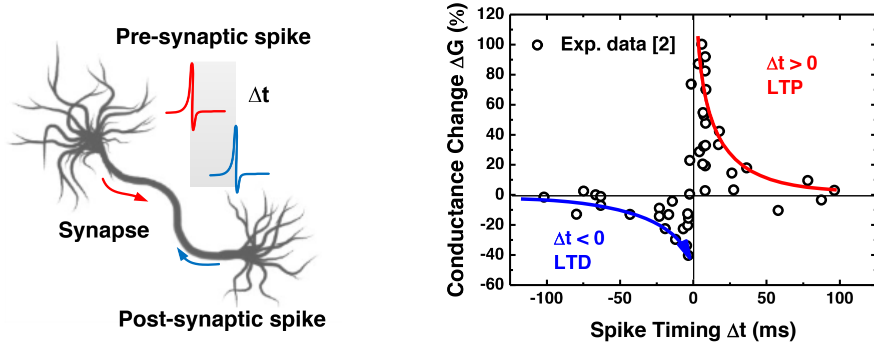
6.1 Introduction

The biophysical neural system has been a rich source of inspiration for computing beyond the conventional von Neumann architecture. By connecting a massive number of spiking neurons through synapses, our brain learns how to recognize various objects and make decisions. It is also hypothesized that training is achieved through plastic synapses, which change their weights based on the spike timing of presynaptic and post-synaptic neuron. This learning rule is known as spike-timing-dependent-plasticity (STDP) Song *et al.* (2000), Bi and Poo (1998) (Fig. 6.1(a)).

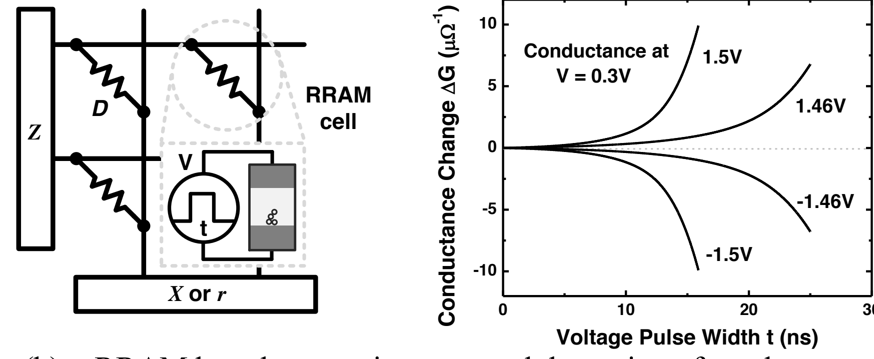
Motivated by neurophysics, sparse coding was successfully developed to pave the way for deep learning with big data Olshausen and Field (1996) Tosic and Frossard (2011). It aims to minimize the following objective function:

$$\sum ||D.Z_i - x_i||^2 + \lambda |Z_i|_1 \tag{6.1}$$

where where x_i is an input vector, λ is the regularization parameter, D is called the dictionary, and Z_i is the feature vector which is assumed to be sparse. If x has p dimensions, Z has m dimensions ($m > p$), then D forms a $m \times p$ matrix (or a 2-D array). To quickly reach a stable sparse representation for x_i, state-of-the-art algorithms apply iterative, parallel, or stochastic methods for the two most computationally intensive tasks: updating the feature vector Z and updating the dictionary D. In this paper, we focus on the Iterative Shrinking-Thresholding Algorithm (ISTA Daubechies *et al.* (2004)) to update Z due to its inherent parallelism, and the Stochastic Gradient



(a) STDP in a biological synapse.



(b) RRAM based crosspoint array and the tuning of conductance (G).

Figure 6.1: Similarity of biological neural network and the RRAM crosspoint array, in both the network structure and device plasticity. The conductance of RRAM cells can be programmed to particular values using programming voltage of specific pulse width.

Descent (SGD Bottou and Bousquet (2008)) to update D exploiting stochasticity for greater efficiency:

1. **Update Z** via ISTA: $Z_{t+1} \leftarrow h_{\lambda/L}(Z_t - D_t^T.r_t)$, where $h_{\lambda/L}$ is soft thresholding function, and $r_t \triangleq D_t.Z_t - X_t$ is the residual error of data presentation (r).

2. **Update D** via SGD: $D_{t+1} \leftarrow D_t - \eta_t.\Delta D_t$, where η_t is the learning rate and $\Delta D_t = r_t.Z_t^T$.

These learning algorithms are typically implemented in software, and run on a general-purpose CPU/GPU. Limited by the sequential architecture of todays micro-

processors, they suffer from long computing times, especially in dealing with a large D matrix. Thus, it is desirable to have a special hardware that accelerates the learning process beyond such limitations.

The resistive crosspoint array structure, shown in Fig. 6.1(b), was recently proposed as a promising solution for learning in hardware neural networks (Afifi *et al.* (2010), Rajendran *et al.* (2013)). The iterative solution to the sparse coding problem can be realized by mapping the matrix D onto the resistive array, and *learning* takes place through the update step. The quantity X (or r) is associated with one side of the array and Z with the other side. In this way, the crosspoint mimics the structural map of a neural system. At each cross point, the conductance (G) of a memory cell represents the synapse weight. The memory technology of choice is resistive random access memory (RRAM), due to its non-volatility, integration density, and low power consumption (Jo *et al.* (2010)). The inset of Fig. 6.1(b) illustrates its structure. Analogous to a synapse device, G of a RRAM cell is increased (or decreased) by a positive (or negative) voltage pulse. The amount of change depends on the voltage value and the pulse width (Fig. 6.1(b)).

The basic functions of the crosspoint array include:

- **Read for Matrix Product**: When a voltage is input from $Z(V_{Z,j})$, the output current at x_i is $I_{X,i} = \sum G_{i,j}.V_{Z,j}$. If G encodes D, then a Read corresponds to sensing the current which encodes $D.Z$, which takes in parallel.

- **Write to Update D**: The conductance of the entire array is updated in parallel. Previous approaches involve sequential operations (row-by-row, column-by-column, or even bit-by-bit) to update G of the RRAM cell.

However, when these functions are implemented in a monolithic technology, the unusually large dimension of D (i.e., large fan-in and fan-out to each X and Z node)

poses unique challenges to periphery circuit design: for Read, the receiver needs to convert a tremendously wide range of output current I_i (> 100× difference) to a digital data at high precision; for Write, it is preferred to program all cells in parallel for high-speed computation, with local data only from pre-synaptic and post-synaptic nodes, as observed in a biophysical synapse. We present effective solutions to these challenges.

The remainder of the chapter is organized as follows. Section 6.2 describes the parallel architecture and principles of Read and Write circuitries. Section 6.3 presents experimental results from a 65nm CMOS design, and a learning demonstration is shown in Section 6.4. The chapter is concluded in Section 6.5.

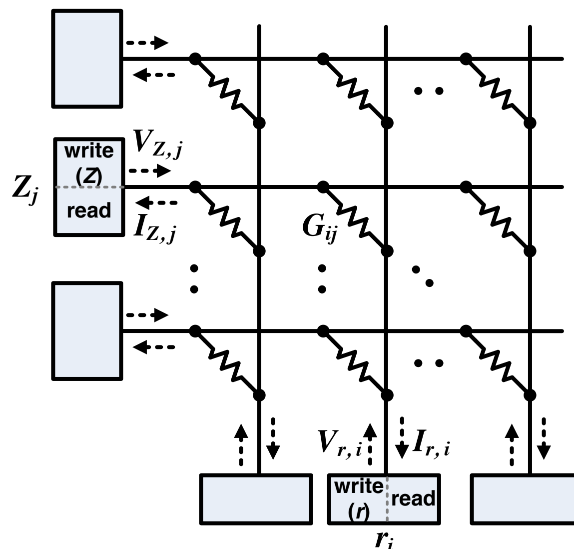


Figure 6.2: PARCA architecture with peripheral Read and Write modules. Z and X (or r) nodes have the same Read (Section 6.2.1), but different Write circuits (Section 6.2.2). All RRAM cells are Read or Written in parallel.

6.2 Crosspoint Array Architecture and Design

Fig. 6.2 illustrates the proposed parallel architecture with resistive crosspoint array (PARCA). The D array connects Z on one side and r on the other side. The two key operations that we intend to fully parallelize are: $D.Z$ and D update.

- $D.Z$ (*or* $D^T.r$): Parallel Read of the RRAM array. For each non-zero bit of Z, a small read voltage is applied simultaneously. The read voltage V_Z is multiplied with G at each crosspoint, and the weighted sum results in the output current at each r node. The read circuitry described in Section 6.2.1 converts this current into a binary number. Compared to conventional memory arrays that require reading row-by-row, our approach reads the entire RRAM array in parallel, without the sneak path problem (Liang *et al.* (2013)) found in the memory application of RRAMs, thereby accelerating $D.Z$. A similar Read operation in the transpose direction computes $D^T.r$.

- D update: Parallel Write of the RRAM array. In SGD, the change of D is proportional to $r.Z$ (Bottou and Bousquet (2008)). By properly generating voltages at local r_i and Z_j nodes, current G_{ij} of a RRAM cell is changed by an amount proportional to $r_i.Z_j$. Thus, all RRAM cells are modified in parallel, achieving considerable speedup compared to previous approaches that require read-modify-write operations. The proposed write circuitry is described in Section 6.2.2. Table 6.1 summarizes the key operations handled by PARCA.

6.2.1 *Read: Integrate and Fire*

The proposed Read circuit is essentially a current-to-digital converter, where it senses the output current at each r_i (or Z_j) node for $D.Z$ (or $D^T.r$), and converts to digital values. In principle, this output response is similar to that of a biological

Table 6.1: PARCA operations for key sparse coding tasks.

Task	PARCA Method
$D.Z$	$I_{r,i} = \sum_i G_{ij}.V_{Z,j}$
$D^T.r$	$I_{Z,j} = \sum_j G_{ij}.V_{r,ij}$
D update	$\Delta G_{ij} = \eta.r.Z$
Read	Input: small V pulse; Output: I to digital
Write	Input: large V_r and V_Z pulses, with proper timing between them

neuron model, namely Integrate-and-Fire (IF) (Abbott (1999)). Starting from a reset voltage, the output current is integrated on the finite capacitance of each RRAM column; when the voltage charges up above a certain threshold, the output switches and the capacitance is discharged back to the reset voltage. The read property of a RRAM cell further poses a constraint that the reset voltage and the threshold voltage should be very close to each other; otherwise the output current does not represent the correct weighted sum (Wong *et al.* (2012), Yu *et al.* (2013)). In our 65nm design (Section III), the reset voltage and the threshold voltage are 500mV and 530mV, respectively. To meet this constraint, an asynchronous comparator with high sensitivity to small changes in voltages was required, and we employed an adaptive Schmitt trigger to create the IF neuron circuit (Wang (1991)).

For $D.Z$, we measure the integrated current at each r_i node by counting the number of times (n_i) the voltage at the integration node crosses the set threshold within a read timing window (T_R). As the charge accumulates over time on a finite capacitance, the time it takes for the integration voltage to exceed the threshold is inversely proportional to the current ($I.t = \text{constant}$). Since $n_i \propto 1/t$, the current will be proportional to the number of spikes that occurred during a fixed timing window. Fig. 6.3 shows the Read circuit where the capacitance used to integrate the current is

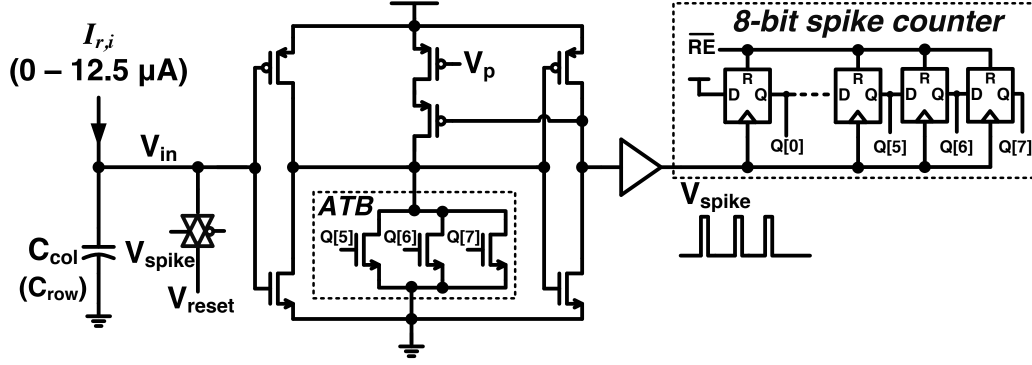


Figure 6.3: Circuit schematics of the Read circuit. Based on the IF neuron model, it converts a wide range of input current $I_{r,i}$ into a digital number.

the parasitic capacitance of the RRAM column or row. The transmission gate (TG) discharges the capacitance while the adaptive threshold block (ATB) strengthens the pull down network to vary the threshold below 530mV only when the incoming current is high. The output of the Schmitt trigger is buffered and drives the clock input of a 8-bit shift register to store n_i.

6.2.2 Write: Timing based Local Programming

To change the conductance of an RRAM cell, the voltage across the cell should be Vdd, and $Vdd/2$ only induces negligible change on G due to its strong dependence on the voltage Liang *et al.* (2013). Inspired by STDP in a biological neuron, G is programmed by the overlap time between local r and Z signals: The write circuit for Z generates a pulse with a duty cycle proportional to Z, while a spike train is generated at r with the firing rate proportional to r and the pulse width is fixed at 1ns. Wherever the pulse at r is overlapped with Z, it creates $|V_Z - V_r| = Vdd$. Therefore, the total programming time equals to the overlap between Z and r, i.e., $r.Z$. Since Z is always positive while r can be positive or negative, we divide the write period into a positive/negative period for $r > 0/r < 0$.

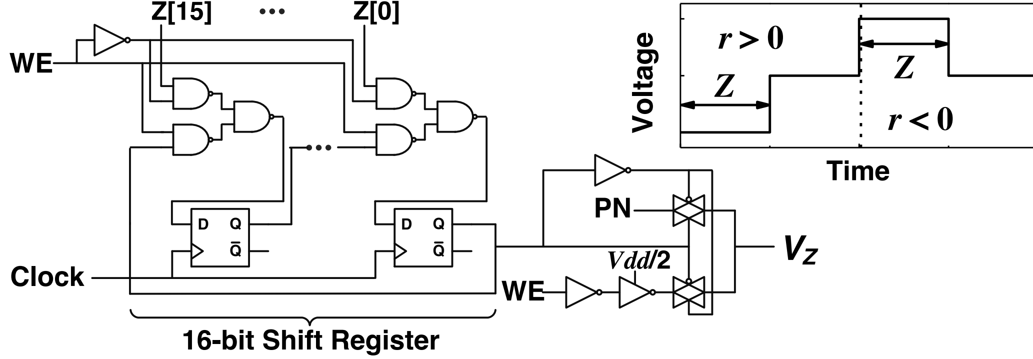


Figure 6.4: Write circuit for Z, with two periods for $r > 0$ and $r < 0$.

Write Circuits for *Z*: In the positive period, Z is 0 for a certain time proportional to Z; then it switches to $Vdd/2$. The overlap time between $Z = 0$ and $r = Vdd$ tunes the RRAM conductance. A similar scenario is designed for the negative period, with a symmetric polarity, as shown in Fig. 6.4.

Fig. 6.4 shows the digital design to generate such a pulse pattern. A 16-bit shift register converts the parallel input Z[15:0] into a sequential output. The time when the output is 1 is proportional to the value of Z. The output of the shift register is connected back to the first stage of itself in order to recycle the data Z. With 32 clock cycles for one write period, it generates two identical pulses with the duty cycle proportional to the value of Z. These two identical pulses are connected to multiplexors to generate different programming voltages for the positive period and the negative period.

Write Circuits for *r*: The train of pulses generated at r has its pulse number proportional to the value of r, where each pulse has a fixed width for fixed RRAM programming period. The pulses are evenly distributed across the write period in order to minimize the quantization error.

Fig. 6.5 presents the design for generating the r signal. It consists of various delay elements forming a configurable ring oscillator (RO) with the polarity control by the

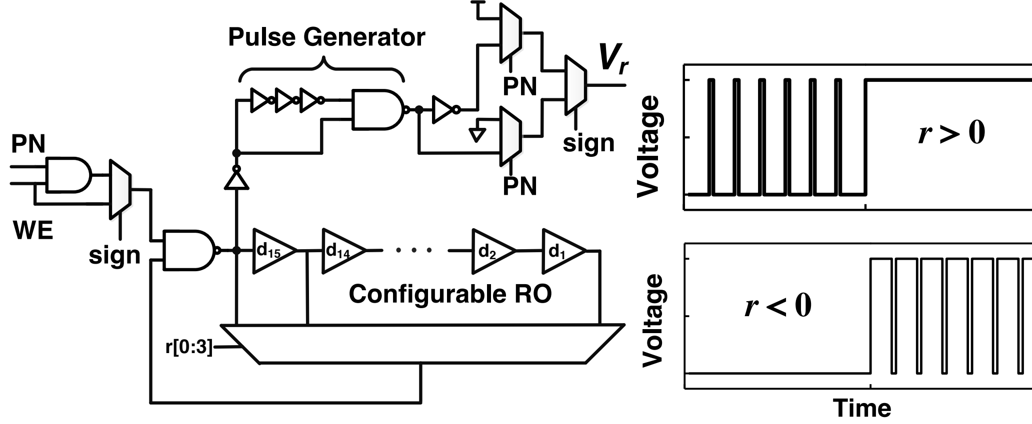


Figure 6.5: Write circuit for r, with the firing rate proportional to r.

sign-bit of r. The number of pulses during the write period (i.e., the firing rate) is varied by adding switches into the oscillation loop, which determine the total gate delay in the ring oscillator. The control signal of the switches is generated from the r value, ensuring that only one switch is on for a particular value of r. When $r = 0$, no change in the RRAM conductance is allowed. In total, 15 buffer stages ($d_1 - d_{15}$) are implemented with different delay values, such that the number of pulses generated in each write cycle is proportional to the r value. From each rising edge of the RO output, the pulse generator generates a pulse with fixed 1ns width. This ensures the total programming time is proportional to the pulse number for our RRAM technology. The sign-bit of r and the write phase (PN) finally select the output signal among Vdd, 0, pulse generator output or the inversion of it.

6.3 65nm CMOS Implementation

The read and write circuitries are implemented in 65nm CMOS technology. These circuits are simulated with the RRAM model Yu *et al.* (2013) that is calibrated with measurements.

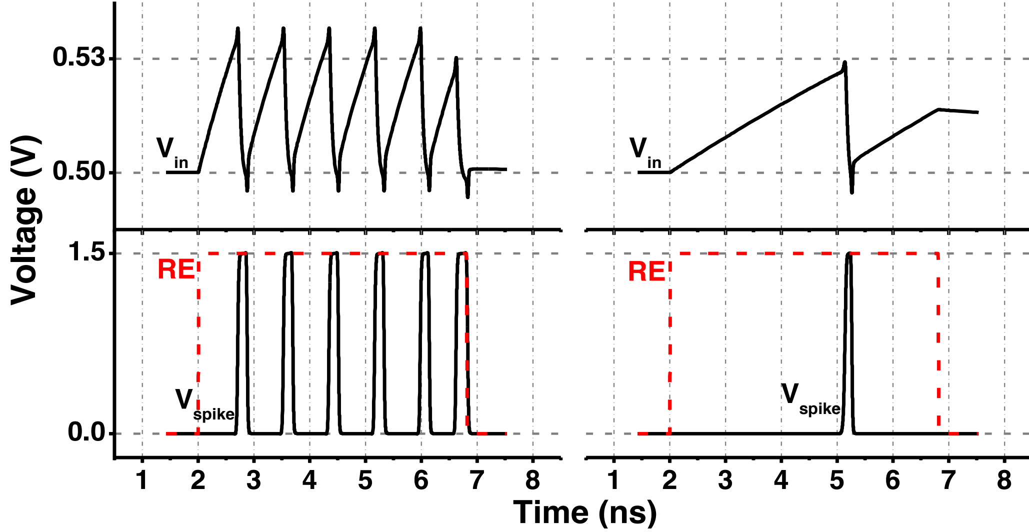


Figure 6.6: The operation of the read circuit for two input current: (left) $I_r = 6.5\mu A$; and (right) $I_r = 1.1\mu A$; the corresponding n_i is 6 and 1.

6.3.1 Read

Fig. 6.6 demonstrates the proper operation of the read circuit, with two values of input current. The RRAM current integrates at the input node (V_{in}), increasing the voltage until it reaches the threshold of the Schmitt trigger. The circuit then initiates reset to discharge the capacitance. This integrate-reset process continues while Read Enable (RE) is high. The number of reset pulses (n_i) present in this timing window (4.6 ns in our design) is recorded by enabling the shift register for each reset pulse.

As shown in Fig. 6.8(a), the number of reset pulses linearly increases with incoming RRAM current at $\sim 1\mu A$ granularity. Non-linearity exists at high G values, which is due to the finite discharge time of the capacitance and the voltage overshoot above threshold due to latency. The non-linearity further limits the lower bound of the read time window, forcing a longer read time. Therefore, we introduce the ATB unit, which is only enabled when the conductance is high, to ensure high linearity between G and η_i , as demonstrated in Fig. 6.8(a).

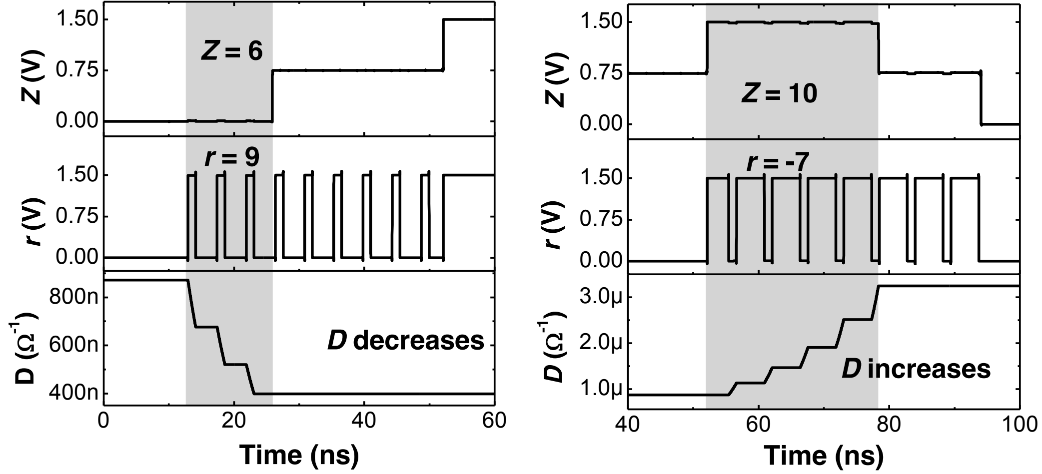


Figure 6.7: The overlap in time between Z and r pulses tunes D.

6.3.2 Write

Fig. 6.7 shows the timing diagram of the parallel programming system with programming time of 84 ns. When the write enable (WE) signal turns on, both Z and r write circuitries start generating the pulses based on the values of Z and r and, thus change the value of D during the overlap time. Fig. 6.7 demonstrates that when r is positive, the programming occurs in the positive period and the value of D decreases; when r is negative, the programming happens in the negative period and the value of D increases.

The method of using overlap time of Z and r pulses with a certain granularity to calculate $r.Z$ introduces quantization error. To analyze this, we performed simulations for all Z and r values. Fig. 6.8(b) compares the simulated results to an ideal multiplication. The digital programming closely follows the theoretical value, while producing the maximum error of 1 bit (out of 16 bits) when both Z and r are small.

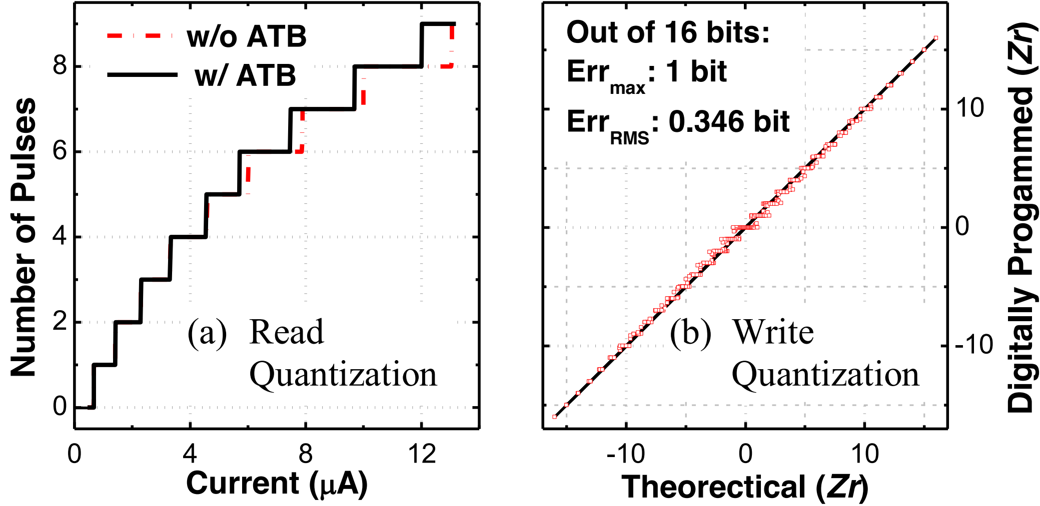


Figure 6.8: Quantization of read and write circuits are shown. (a) Number of pulses and RRAM current show a close-to-linear relationship. (b) Digitally programmed pulse width closely follows the mathematical multiplication.

6.4 Demonstration in Learning

We demonstrate the proposed system on the task of sparse coding, and compare it against a software implementation. MNIST data set LeCun *et al.* (2010) is used to learn the dictionary and extract the image features, with ISTA Daubechies *et al.* (2004) and SGD algorithm Bottou and Bousquet (2008). The software ran on an Intel Core i7 3.4 GHz 8-core processor. The initial dictionary and the learned dictionary are shown in Fig. 6.9. It can be seen that the learned dictionary well captures local features. Table 6.2 summarizes the computation time and energy consumed by the software and our PARCA system. Both steps of Update Z and Update D benefit from the parallel computing of $D.Z$ ($D^T.r$); Update D is further accelerated by the parallel write of . Overall, PARCA achieves more than 3000× speedup over the software implementation, with higher power efficiency.

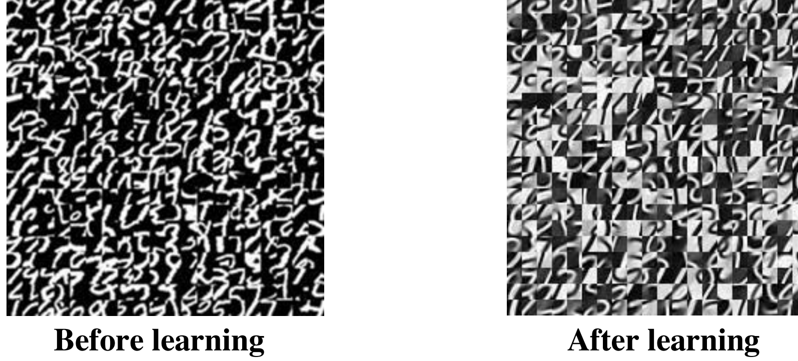


Figure 6.9: Demonstration of dictionary learning with MNIST data.

Table 6.2: Evaluation of the speedup in computing and energy.

Task	Software of CPU	PARCA	Acceleration
Update Z	17.2 ms (matrix op.)	5 μs (200 Read)	3440X
Update D	26.4 μs (matrix op.)	84 ns (1 Write)	314X
Total Time	17.2 ms	5.01 μs	3430X
Total Energy	208 mJ	0.2 J	-

6.5 Conclusion

In this paper, we proposed a parallel architecture with resistive crosspoint array for dictionary learning applications, where each dictionary value is represented by the conductance of a RRAM cell. The proposed bio-inspired read circuit converts the RRAM current into digital values in an integrateand-fire fashion. Analogous to STDP, the write circuits employ local signals of Z (duty cycle) and r (number of pulses or spikes) to update the conductance of the entire RRAM array in parallel. Peripheral circuits were implemented in 65nm CMOS, and simulated with RRAM device models to accelerate computation-intensive tasks in dictionary learning. PARCA demonstrates 3000X acceleration for an image feature extraction task when compared to ISTA and SGD sparse coding software.

Chapter 7

RANDOM SPARSE ADAPTATION

An array of multi-level resistive memory devices (RRAMs) can speed up the computation of deep learning algorithms. However, when a pre-trained model is programmed to a real RRAM array for inference, its accuracy degrades due to many non-idealities, such as variations, quantization error, and stuck-at faults. A conventional solution involves multiple read-verify-write (R-V-W) for each RRAM cell, costing a long time because of the slow Write speed and cell-by-cell compensation. In this chapter, we shall look at a fundamentally new approach to overcome this issue: random sparse adaptation (RSA) after the model is transferred to the RRAM array. By randomly selecting a small portion of model parameters and mapping them to on-chip memory for further training, we demonstrate an efficient and fast method to recover the accuracy: in CNNs for MNIST and CIFAR-10, 5% of model parameters is sufficient for RSA even under excessive RRAM variations. As the back-propagation in training is only applied to RSA cells and there is no need of any Write operation on RRAM, the proposed RSA achieves 10-100X acceleration compared to R-V-W. Therefore, this hybrid solution with a large, inaccurate RRAM array and a small, accurate on-chip memory array promises both area efficiency and inference accuracy.

7.1 Introduction

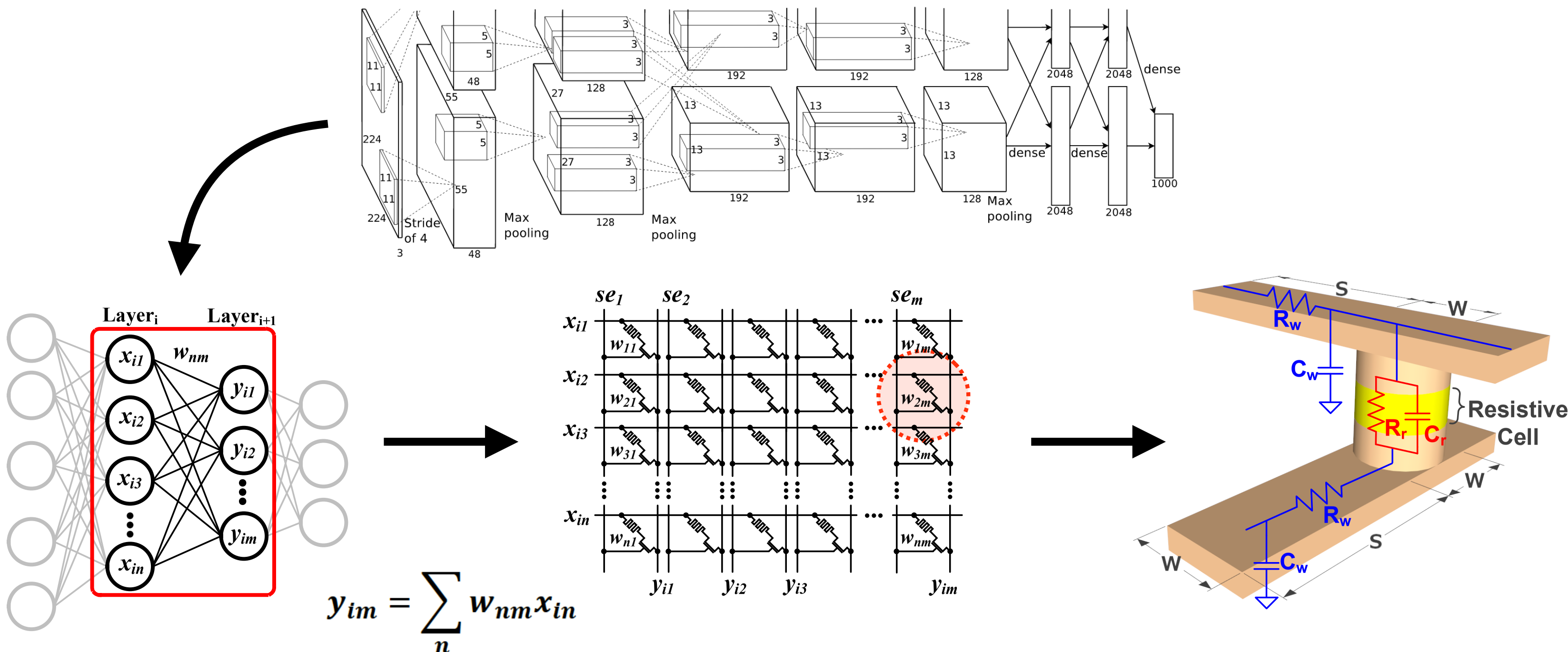
Recent years have witnessed dramatic advances in deep learning research. These networks involve multiple layers with the previous layer feeding the next layer (Fig. 7.1). However, in order to achieve human-level accuracy or even better, they tend to be very complicated and demand a large amount of computation resource. The

need of hardware acceleration has been urgent and kindled a high interest on new architectures and emerging devices. Among them, multi-level RRAM devices have demonstrated the potential to speed up the dot product of vector-matrix multiplication (Fig. 7.1) (Xia *et al.* (2016)), achieving high energy efficiency and small footprint (Gao *et al.* (2015), Alibart *et al.* (2012), Lee *et al.* (2012)).

On the other side, a realistic RRAM array has many non-idealities: high device-to-device variation, limited precision, stuck-at-faults, limited on/off ratio, etc. A Closed-Loop-on-Device (CLD) scheme was used in Hu *et al.* (2013), which repeatedly performs programming and sensing to do gradient descent on-chip. However, it has an expensive feedback control and multiple writes to the RRAM, which is time consuming. Other works like Liu *et al.* (2014), explored the Open-Loop-off-Device (OLD) scheme, where pre-trained models are used to calculate the resistance values of the devices and then programming and sensing is done over loop, Read-Verify-Write (R-V-W), till the resistance values converge to the desired values. Variation-aware training schemes are used in Liu *et al.* (2015a) and Chen *et al.* (2017), where read operations are first done on the array to characterize the devices and model the device variations in the array. This model is then used as an input while training the neural network offline. The drawback of this method is that the neural network has to be trained from scratch for each chip. Moreover, inference accuracy in all of the above methods are limited by the quantization error (number of levels) of the RRAM device. Recognizing the expensive cost in time and characterization of RRAM, this work proposed a novel scheme, with the contribution on:

1. Quantitative analysis of various non-idealities in the RRAM array on inference accuracy of two representative datasets, MNIST and CIFAR-10.

2. A fundamentally new approach, Random Sparse Adaptation (RSA), to mitigate



$$y_{im} = \sum_n w_{nm} x_{in}$$

Figure 7.1: Each layer in a deep neural network can be mapped to a RRAM array for acceleration. The neural network is first trained offline and the optimized model paramters are selected. The weights of connections in neural network are then encoded as conductance values of RRAM devices using write pulses of appropriate pulse widths. Layerwise execution can be performed in parallel given that we have enough read circuits to collect and digitize column outputs.

the impact with high effectiveness and efficiency.

3. Elimination of Write or device-level characterization to recover accuracy under RRAM non-idealities. RSA achieves 10-100X speedup compared to R-V-W.

4. The hybrid implementation of RSA using a parallel, small, high precision on-chip memory with the main, large, inaccurate RRAM array, enhancing the accuracy by > 10 % for CIFAR-10 using RRAM only.

7.2 Non-ideal effects in a RRAM device

A realistic RRAM device only has finite levels, limited by On/Off resistance, variations, etc. Write variation is assumed to follow a lognormal distribution (Lee *et al.* (2012)):

$$r' \leftarrow e^{\theta}.r \tag{7.1}$$

$$\theta \sim \mathcal{N}(0, \sigma) \tag{7.2}$$

where, r' is the actual value programmed, r is the intended value, $\mathcal{N}$ is the normal distribution with 0 mean and σ standard deviation (Table 7.1). The nominal *on-* and *off-* state resistances are set to $10k\Omega$ to $1M\Omega$ in our study. We assume 32 levels, even though Alibart *et al.* (2012) demonstrated reliable operations with 128 levels. Stuck-at-faults occur when a device is always at either high resistance state (SF1) or low resistance state (SF0). In an array, SF1 and SF0 are assumed to affect 9.04% and 1.75% of the devices, respectively (Chen *et al.* (2015b)). Cycle-to-cycle variation (read variations) occur due to random noise in CMOS periphery circuits. They are negligible compared to write variations and can be mitigated by improving CMOS circuit design.

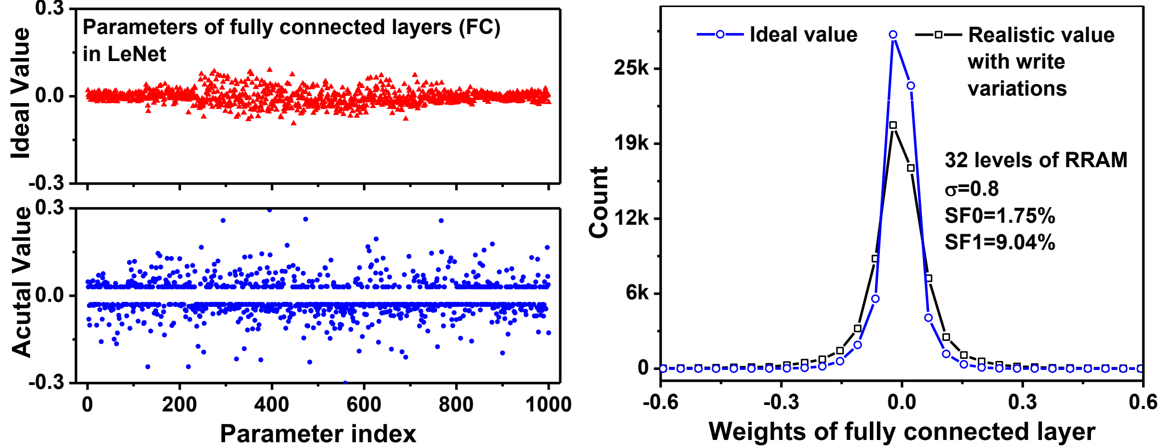


Figure 7.2: The deviation of model parameters after write to the realistic RRAM array. While the pre-trained models are 32-bit floating point numbers, real RRAM devices can have 32 level of quantization. This distorts the distribution of model parameters by forcing pretrained weights to nearest quantization level. Also RRAM cannot encode the value of 0, because the conductance value cannot be 0. As a result all parameters close to 0 are forced to the minimum value that can be encoded by the RRAM (based on the maximum resistance state). That results in a step near values of 0 as shown here.

Due to these non-idealities, the distribution of pre-trained model parameters is distorted when they are programmed to a RRAM array (Fig. 7.2), resulting in significant degradation of inference accuracy (Fig. 7.3). The performance of more complex datasets, such as CIFAR-10, is even more sensitive to these device effects, as illustrated in Fig. 7.3. Further study in Fig. 7.4 and Fig. 7.5 confirmed that device-to-device variation and the quantization level affect the accuracy the most.

7.3 Random Sparse Adaptation

To recover the accuracy loss, R-V-W is commonly used. However, device-level R-V-W is not effective: even R-V-W of the top ranked parameters requires a large

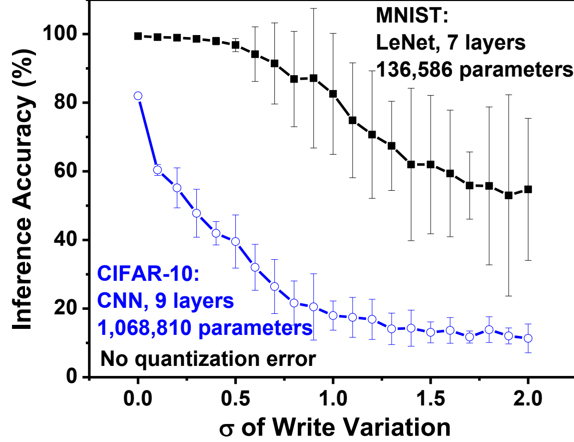


Figure 7.3: Accuracy degradation due to device non-idealities in the form of write variations, for two representative convolutional neural networks (LeNet for MNIST dataset and a 9 layered CNN for CIFAR-10). It is assumed that the write variation follows normal distribution. As demonstrated, deeper networks for more complicated tasks are affected to much greater extent. When write variations have $\sigma \geq 1.0$, the outputs from RRAM array are almost random.

Table 7.1: Assumptions of major types of RRAM device non-idealities. Write variations is considered to follow a normal distribution with mean at the desired value. Stuck-at-high (SF1) arises when certain cells are always at low impedance state no matter what value in written to it. Similarly, Stuck-at-low (SF0) are cells which are always at high impedance state irrespective of the value written to them.

Parameter	Values
Write variation	$r' \rightarrow re^{\theta}$, $\theta \sim \mathcal{N}(0, \sigma)$
Quantization level	32
Stuck-at-high (SF1)	1.75%
Stuck-at-low (SF0)	9.04%

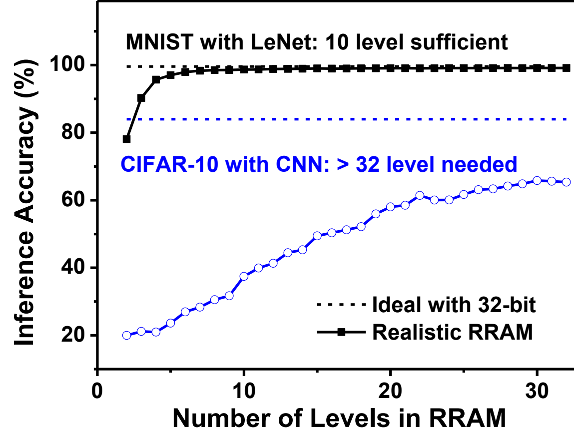


Figure 7.4: Effect of limited quantization levels available in RRAM devices. It is assumed here that the devices have no write variations. As shown, number of levels is critical to inference, especially for complicated tasks and deeper neural networks. In this work we have assumed 32 levels in RRAM devices.

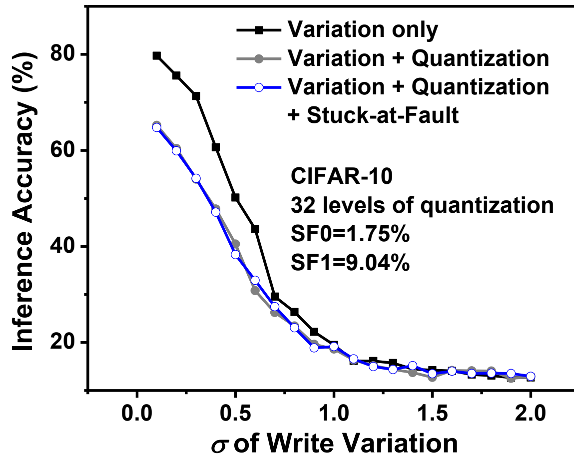


Figure 7.5: Effect of all non-idealities in RRAM arrays on inference accuracy. The critical non-idealities include device-to-device write variations, quantization errors, stuck-at-faults (SF0 and SF1). The effects of cycle-to-cycle read variations in RRAM devices is negligible compared to others and is not considered in this work. Write variation and quantization have the most significant impact on accuracy.

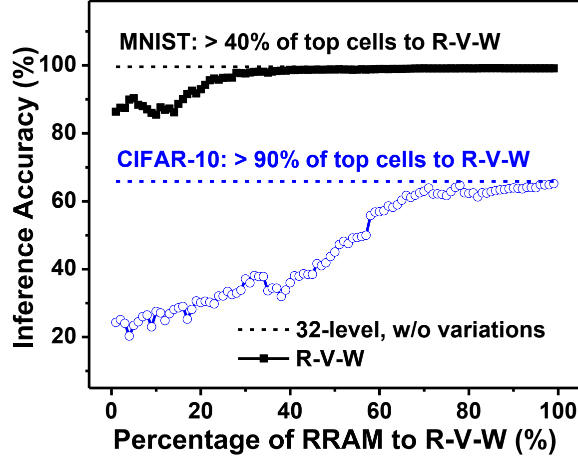


Figure 7.6: A large amount of cells needs to be verified, even if their values are ranked.

portion of them (Fig. 7.6). As accurate write operation in RRAM is very slow compared to on-chip memory (Table 7.2), R-V-W takes a very long time for sufficient accuracy recovery (Table 7.3).

In contrast to device-level R-V-W, machine learning algorithms themselves, such as training, are very robust to parameter changes, because of the redundancy in the solution space and the adaption in training. Such observation inspires us to the RSA scheme for post-model mapping to the RRAM array. Instead of correcting each

Table 7.2: Timing parameters and sizes for RRAM and on-chip memory. On-chip memory, such as Register File (RF), is much faster in Write, but has a larger size.

	RRAM					**RF**
Material	HfOx	TiOx	TaOx	W/Al/PCMO/Pt	AlOx/HfO2	Si
Levels	16	128	4	2	2	32-bit
Write time	6.5s	60s	1.5ms	500s	4 ms	1 ns
Read time	24 404 ns					1 ns
Array size	1000 x 1000					100 cells
Area	$1064F^2$					$1041.5F^2$

Table 7.3: High cost in operation time when R-V-W is applied. This is due to both the long Write time of RRAM devices and the ineffectiveness of R-V-W, even though in R-V-W the parameters are sorted first by their values and top ones are verified. As observed, for MNIST, to recover inference accuracy within 1% of maximum achievable with 32 level RRAM devices R-V-W needs to correctly program top 40% of the parameters which takes $\sim$ 82 seconds. For CIFAR-10, even with correct programming of 100% of the RRAM cells, we can reach accuracy of 65.18% and it takes $\sim$ 2389 seconds. This shows that, verifying and correctly programming every cell to encode desired conductance values is very in-efficient.

% of RRAM cells for R-V-W		5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
MNIST	Accuracy(%)	90.33	85.49	92.98	97.64	98.66	98.79	98.91	99.12	99.12	99.12	99.14
	Time (s)	10.2	20.5	41.0	61.5	82.0	102	123	143	164	184	203
CIFAR-10	Accuracy(%)	23.42	27.55	30.16	37.11	35.99	45.05	56.9	62.95	62.29	63.92	65.18
	Time (s)	121	241	483	724	965	1206	1447	1689	1930	2171	2389

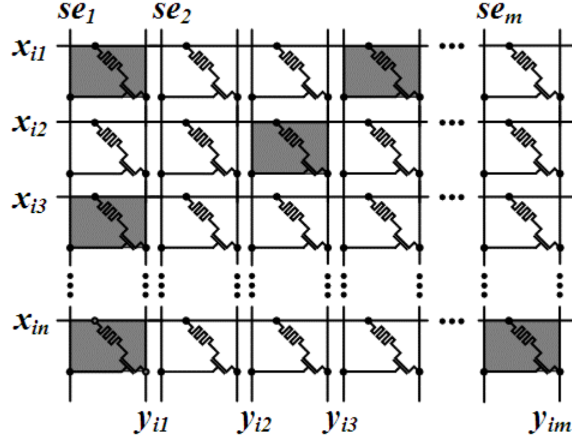


Figure 7.7: RSA randomly selects a certain portion of cells (shadowed cells) and re-trains them. Re-training can be considered as a online learning/adaptation procedure which aims at mitigating the effects of array non-idealities. During the training process, the weights stored on the selected cells are adjusted so as to move the networks transform function to a nearby minimal loss point and thus recover from the lost accuracy.

RRAM cell, what we need to adapt is only a small portion of the model to gain the accuracy back. The selection should be random to cover the feature space; it will be sparse since the majority of the model distribution is still correct. Fig. 7.7 presents the concept of RSA.

7.3.1 Regularized random sparse selection

Running gradient descent on-chip while modifying the RRAM cells is slow and inefficient. The RSA method proposes to randomly select a small subset of cells and replicate them in a separate on-chip memory (Fig. 7.7), effectively enhancing the programming speed in adaptation. The random selection is further regularized in the sense that an equal number of cells will be selected from each row and column of the original array, such that the selected cells can be compiled into a rectangular array

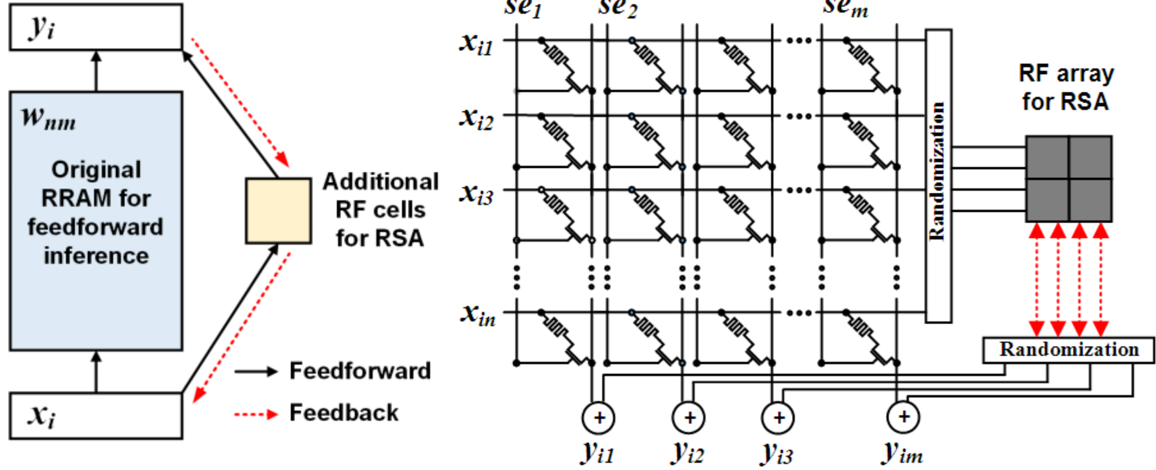


Figure 7.8: The network structure, design and flow using RF for RSA cells. The pre-trained network is first mapped to the RRAM array. The random connections selected for RSA are mapped to RF cell array and initialized with random normal distribution. RSA backpropagates only goes through RF cells. Since we READ/WRITE on RF cells and RRAM is used in read only mode, RSA is very fast.

for a compact footprint of the RSA array and the periphery circuitry (Fig. 7.7). The cell positions are still random and the random connection between RSA and RRAM input/output are hard wired (Fig. 7.8). Pseudo code for RSA on hardware is shown in Algorithm 5.

7.3.2 Network adaptation using RSA

Fig. 7.8 presents the architecture, design and operation of the proposed RSA scheme. First, pre-trained models are programmed to RRAM array and the on-chip RSA cells are initialized from random-normal distribution with 0 mean and 0.1 standard deviation. During the feedforward path for inference, the input to the layers are passed to both RRAM array and parallel on-chip RSA memory. The output from both of these are added to generate the overall layer output. During back-propagation

Algorithm 5 Pseudo code for Random Sparse Adaptation on hardware.

```
1: procedure RSA(Network architecture Net_ideal, Dataset)
2:     Train baseline: W_ideal ← train(Net_ideal, Dataset)
3:     Add RRAM models: W_real ← addVariations(W_ideal)
4:     for conv and fc layers in Net do
5:         Create parallel RSA trainable layers
6:         Initialize: W_RSA ← truncated normal distribution
7:         Create mask: mask; W_RSA ← W_RSA × mask
8:     while convergence ≠ True do
9:         Forward: read RRAM and RSA
10:        Compute gradient: gr ← gradients()
11:        Remove gradients of not selected: gr_masked ← gr × mask
12:        Calculate weight change: D_WRSA ← f(gr_masked)
13:        Backpropagation: write RSA only, W_RSA ← W_RSA + ΔW_RSA
```

for training, RRAM only has Read and the output is combined with RSAs to calculate the gradient, which is then used to adapt the RSA cells. As the parameters on RRAM array are masked as non-trainable in this method, no Write is applied on RRAM cells. Only the on-chip RSA cells are modified to tune the overall network in the direction to improve the accuracy. Fig. 7.8 shows the pseudo code for training the on-chip RSA parameters. With the elimination of Write on RRAM, the speed of this method is not limited by RRAM device anymore.

7.4 Demonstration of RSA

To demonstrate the efficacy of RSA, two representative datasets are used, MNIST for handwritten digit recognition and CIFAR-10 for more complicated image recognition. RRAM Write variations, quantization error and stuck-at-faults are modeled in Table 7.1. To estimate the performance of R-V-W and RSA, related timing pa-

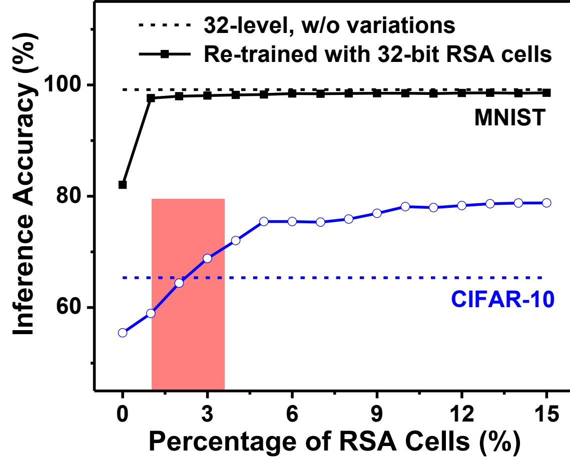


Figure 7.9: A small number of 32-bit RSA cells effectively improves the accuracy. As observed, for simple tasks like MNIST, only $3 \sim 5\%$ of total connections in RSA can push the accuracy back to software baseline. An interesting observation here is that, for complicated tasks like CIFAR-10, RSA can push accuracy to much higher values than what can be obtained with a stand alone ideal RRAM array. Since the RSA cells are 32-bit floating point numbers, they mitigate the effects of 32-level quantization to a great extent and thus achieve higher accuracy.

rameters are summarized in Table 7.2. 32-bit register files (RF) are assumed to be the on-chip memory for RSA because of the flexibility. We further assume that an array can have a maximum of 16 read circuits; when the array size is large, multiple columns will share read circuits. For back-propagation, the CMOS circuits calculating pooling and gradient descent are assumed to be outside of the RRAM and RSA array, with enough number of processing elements (e.g. 16 cores) to complete the calculation in time. Overall, the performance of R-V-W and RSA is limited by the Write time of RRAM and the Read time of RRAM, respectively. Due to the small size of RSA cells and their fast speed, the parallel path on RSA is not the critical path.

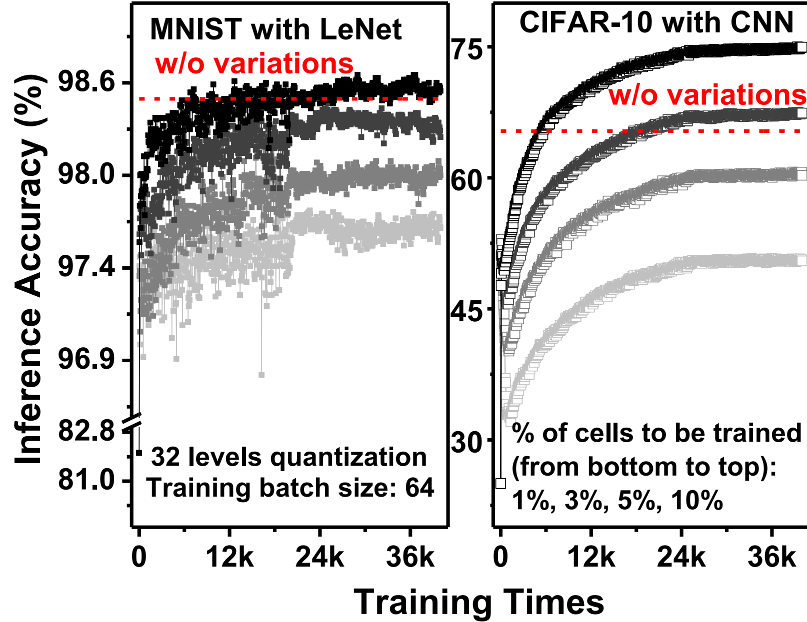


Figure 7.10: The training of RSA does not require the full dataset to recover the accuracy. With increase in the number of RSA cells, the number of training iterations gets reduced. This behaviour is expected because with few RSA cells, the optimization algorithm (SGD) has to move the values to greater distance to reach the minima as the degrees of freedom available to SGD is less. So it needs more iterations of weight updates.

Fig. 7.9 shows that using RSA, only a very small portion of parameters ($< 5\%$) is required to compensate the effect of device variations and stuck-at-faults on inference accuracy. With gradient descent operating over high-precision RF cells, a much better accuracy can be achieved than that with 32-level RRAM only. For instance, in the 9-layer CNN for CIFAR-10, the addition of 5% RSA cells boosts the accuracy by more than 10%, which is very significant for this task. Moreover, Fig. 7.10 demonstrates that the training of RSA only require an additional 15% of the training iterations compared to that for the original RRAM. As the percentage of RSA cells increase, the iteration time can be further reduced.

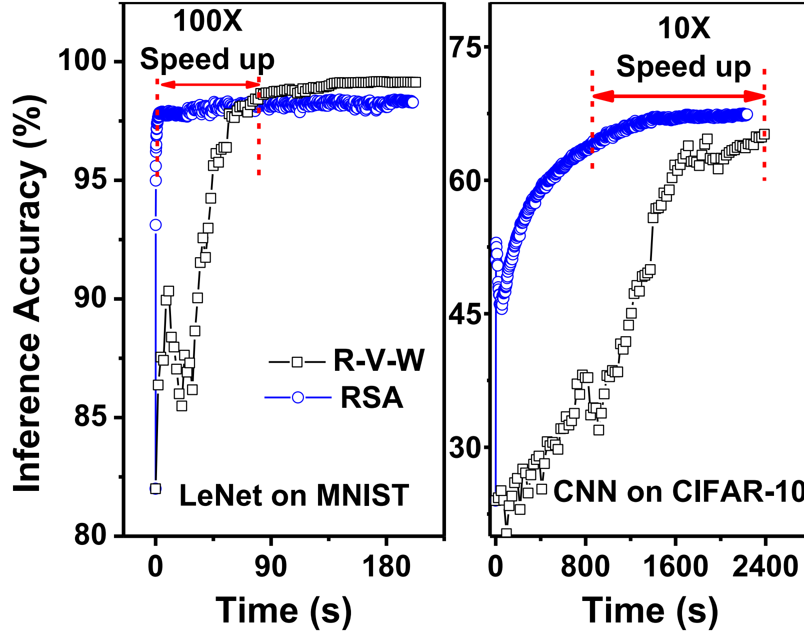


Figure 7.11: RSA rapidly recovers the accuracy, achieving $10 - 100\times$ speedup over R-V-W. With writes to RF cells only, RSA removes the need for slow accurate RRAM writes.

Finally, Fig. 7.11 compares the improvement in accuracy and the time needed, between RSA and R-V-W. Leveraging the robustness of the algorithm, rather than device-level precision, RSA achieves higher accuracies at a much faster speed. The speed-up is in the range of 10-100X, depending on the Read/Write time of RRAM devices and the number of convolutions in the algorithm.

7.5 Conclusions

RRAM based computing has a great potential towards power-efficient hardware acceleration of deep learning algorithms. One of the bottlenecks are non-ideal device effects, especially variations and quantization errors. Previous methods involve looped R-V-W at the device level and are inefficient in practice. Inspired by the intrinsic robustness of machine learning algorithms, this work proposes a novel on-chip training

scheme by randomly selecting a small portion of model parameters, mapping them to a parallel bank of on-chip memory, and adapting them after model mapping to the hardware. The RSA method completely removes the need of Write on RRAM after mapping. As demonstrated on MNIST and CIFAR-10, $< 5\%$ parameters need to be selected as RSA cells under $> 30\%$ variations, achieving 10-100X acceleration to R-V-W. The integration of RRAM and on-chip memory in RSA further offers the operation flexibility and high accuracy beyond RRAM only approaches.

Chapter 8

SUMMARY

This work presents a comprehensive study of deep learning algorithm developement and hardware acceleration to achieve efficient real-time performance. we demonstrate the need and criticality of hardware-software co-optimization for efficient execution of deep learning. We demonstrate this with implementation and optimization strategies at various stages of DNN algorithm and hardware development. The main contributions of this work are:

1. **High performance of deep learning**: We implement a deep neural network from scratch for automatic cough detection from audio data. With the proposed pre-processing scheme and neural network architecture, we were able to achieve state-of-the-art accuracy for cough detection out-performing methods based on traditional algorithms like PCA. Our proposed algorithm achieved 92.3% leave-one-out accuracy on VitaloJAK data captured in real world.

2. **Hardware acceleration using FPGAs**: We implement hardware accelerators for deep convolutional neural networks and random forest trees using FPGAs. With our proposed optimization strategies, we demonstrated high throughput and efficient execution of these. For face detection using random forest trees, our proposed accelerator achieved $\sim 30\times$ performance gain compared to CPUs. For deep convolutional neural networks, our optimization schemes achieved 30.9 GOPs and was able to efficiently execute AlexNet and VGG on Stratix V FPGA boards.

3. **Beyond CMOS**: This work also explores emerging architectures like RRAM

crossbars and RRAM arrays to mitigate the bottlenecks associated with CMOS based hardware accelerators. Using our proposed architecture $\sim 3000\times$ performance improvements over CPUs has been demonstrated for online dictionary learning.

4. **Working with non-ideal RRAMS**: This work also examines the realistic RRAM devices and their non-idealities. In this work, we do an in-depth study of the effects of RRAM non-idealities on inference accuracy when a pretrained model is mapped to RRAM based accelerators. To mitigate this issue, we propose Random Sparse Adaptation (RSA), a novel scheme aimed at tuning the model to take care of the faults of the RRAM array on which it is mapped. Our proposed method can achieve inference accuracy much higher than what traditional Read-Verify-Write (R-V-W) method could achieve. RSA can also recover lost inference accuracy $100\times \sim 1000\times$ faster compared to R-V-W.

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: a system for large-scale machine learning.”, in “OSDI”, vol. 16, pp. 265–283 (2016).

Abbott, L. F., “Lapicques introduction of the integrate-and-fire model neuron (1907)”, Brain research bulletin **50**, 5-6, 303–304 (1999).

Abdel-Hamid, O., A.-r. Mohamed, H. Jiang, L. Deng, G. Penn and D. Yu, “Convolutional neural networks for speech recognition”, IEEE/ACM Transactions on audio, speech, and language processing **22**, 10, 1533–1545 (2014).

Abdelfattah, M. S., A. Hagiescu and D. Singh, “Gzip on a chip: High performance lossless data compression on fpgas using opencl”, in “Proceedings of the International Workshop on OpenCL 2013 & 2014”, p. 4 (ACM, 2014).

Afifi, A., A. Ayatollahi, F. Raissi and H. Hajghassem, “Efficient hybrid cmos-nano circuit design for spiking neurons and memristive synapses with stdp”, IEICE transactions on fundamentals of electronics, communications and computer sciences **93**, 9, 1670–1677 (2010).

Alibart, F., L. Gao, B. D. Hoskins and D. B. Strukov, “High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm”, Nanotechnology **23**, 7, 075201 (2012).

Amoh, J. and K. Odame, “Deepcough: A deep convolutional neural network in a wearable cough detection system”, in “Biomedical Circuits and Systems Conference (BioCAS), 2015 IEEE”, pp. 1–4 (IEEE, 2015).

Amoh, J. and K. Odame, “Deep neural networks for identifying cough sounds”, IEEE transactions on biomedical circuits and systems **10**, 5, 1003–1011 (2016).

Avati, A., K. Jung, S. Harman, L. Downing, A. Ng and N. H. Shah, “Improving palliative care with deep learning”, in “Bioinformatics and Biomedicine (BIBM), 2017 IEEE International Conference on”, pp. 311–316 (IEEE, 2017).

Barros, P., S. Magg, C. Weber and S. Wermter, “A multichannel convolutional neural network for hand posture recognition”, in “International Conference on Artificial Neural Networks”, pp. 403–410 (Springer, 2014).

Barry, S. J., A. D. Dane, A. H. Morice and A. D. Walmsley, “The automatic recognition and counting of cough”, Cough **2**, 1, 8 (2006).

Barton, A., P. Gaydecki, K. Holt and J. A. Smith, “Data reduction for cough studies using distribution of audio frequency content”, Cough **8**, 1, 12 (2012).

Bekkerman, R., M. Bilenko and J. Langford, *Scaling up machine learning: Parallel and distributed approaches* (Cambridge University Press, 2011).

Bi, G.-q. and M.-m. Poo, “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type”, Journal of neuroscience **18**, 24, 10464–10472 (1998).

Birring, S., T. Fleming, S. Matos, A. Raj, D. Evans and I. Pavord, “The leicester cough monitor: preliminary validation of an automated cough detection system in chronic cough”, European Respiratory Journal **31**, 5, 1013–1018 (2008).

Birring, S., B. Prudon, A. Carr, S. Singh, M. Morgan and I. Pavord, “Development of a symptom specific health status measure for patients with chronic cough: Leicester cough questionnaire (lcq)”, Thorax **58**, 4, 339–343 (2003).

Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, “End to end learning for self-driving cars”, arXiv preprint arXiv:1604.07316 (2016).

Bottou, L. and O. Bousquet, “The tradeoffs of large scale learning”, in “Advances in neural information processing systems”, pp. 161–168 (2008).

Boureau, Y.-L., J. Ponce and Y. LeCun, “A theoretical analysis of feature pooling in visual recognition”, in “Proceedings of the 27th international conference on machine learning (ICML-10)”, pp. 111–118 (2010).

Canziani, A., A. Paszke and E. Culurciello, “An analysis of deep neural network models for practical applications”, arXiv preprint arXiv:1605.07678 (2016).

Cavalcante, R. C., R. C. Brasileiro, V. L. Souza, J. P. Nobrega and A. L. Oliveira, “Computational intelligence and financial markets: A survey and future directions”, Expert Systems with Applications **55**, 194–211 (2016).

Chang, O., P. Constante, A. Gordon and M. Singana, “A novel deep neural network that uses space-time features for tracking and recognizing a moving object”, Journal of Artificial Intelligence and Soft Computing Research **7**, 2, 125–136 (2017).

Chellapilla, K., S. Puri and P. Simard, “High performance convolutional neural networks for document processing”, in “Tenth International Workshop on Frontiers in Handwriting Recognition”, (Suvisoft, 2006).

Chen, C., A. Seff, A. Kornhauser and J. Xiao, “Deepdriving: Learning affordance for direct perception in autonomous driving”, in “Proceedings of the IEEE International Conference on Computer Vision”, pp. 2722–2730 (2015a).

Chen, C.-Y., H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu and F. T. Chen, “Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme”, IEEE Transactions on Computers **64**, 1, 180–190 (2015b).

Chen, L., J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang and L. Jiang, “Accelerator-friendly neural-network training: learning variations and defects in rram crossbar”, in “Proceedings of the Conference on Design, Automation & Test in Europe”, pp. 19–24 (European Design and Automation Association, 2017).

Chen, P.-Y., D. Kadetotad, Z. Xu, A. Mohanty, B. Lin, J. Ye, S. Vrudhula, J.-s. Seo, Y. Cao and S. Yu, "Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip", in "Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition", pp. 854–859 (EDA Consortium, 2015c).

Chen, X., Y. Wang, X. Liu, M. J. Gales and P. C. Woodland, "Efficient gpu-based training of recurrent neural network language models using spliced sentence bunch", in "Fifteenth Annual Conference of the International Speech Communication Association", (2014).

Chen, Y. and Y. Xue, "A deep learning approach to human activity recognition based on single accelerometer", in "Systems, man, and cybernetics (smc), 2015 ieee international conference on", pp. 1488–1492 (IEEE, 2015).

Chong, E., C. Han and F. C. Park, "Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies", Expert Systems with Applications **83**, 187–205 (2017).

Collobert, R. and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning", in "Proceedings of the 25th international conference on Machine learning", pp. 160–167 (ACM, 2008).

Daubechies, I., M. Defrise and C. De Mol, "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint", Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences **57**, 11, 1413–1457 (2004).

Decalmer, S. C., D. Webster, A. A. Kelsall, K. McGuinness, A. A. Woodcock and J. A. Smith, "Chronic cough: how do cough reflex sensitivity and subjective assessments correlate with objective cough counts during ambulatory monitoring?", Thorax (2006).

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in "Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on", pp. 248–255 (Ieee, 2009).

Dollár, P., Z. Tu, P. Perona and S. Belongie, "Integral channel features", (2009).

Farabet, C., B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision", in "Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on", pp. 109–116 (IEEE, 2011).

Farabet, C., C. Poulet, J. Y. Han and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks", in "Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on", pp. 32–37 (IEEE, 2009).

Farfade, S. S., M. J. Saberian and L.-J. Li, "Multi-view face detection using deep convolutional neural networks", in "Proceedings of the 5th ACM on International Conference on Multimedia Retrieval", pp. 643–650 (ACM, 2015).

Fischer, T. and C. Krauss, "Deep learning with long short-term memory networks for financial market predictions", European Journal of Operational Research **270**, 2, 654–669 (2018).

Gao, L., P.-Y. Chen and S. Yu, "Programming protocol optimization for analog weight tuning in resistive memories", IEEE Electron Device Letters **36**, 11, 1157–1159 (2015).

Gibson, P. G., A. B. Chang, N. J. Glasgow, P. W. Holmes, A. S. Kemp, P. Katelaris, L. I. Landau, S. Mazzone, P. Newcombe, P. Van Asperen *et al.*, "Cicada: Cough in children and adults: Diagnosis and assessment. australian cough guidelines summary statement", Medical Journal of Australia **192**, 5, 265–271 (2010).

Girshick, R., "Fast r-cnn", in "Proceedings of the IEEE international conference on computer vision", pp. 1440–1448 (2015).

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville and Y. Bengio, "Maxout networks", arXiv preprint arXiv:1302.4389 (2013).

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville and Y. Bengio, "Maxout networks (2013)", arXiv preprint arXiv:1302.4389 (2017).

Gupta, S., A. Agrawal, K. Gopalakrishnan and P. Narayanan, "Deep learning with limited numerical precision", in "International Conference on Machine Learning", pp. 1737–1746 (2015).

Han, S., X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network", in "Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on", pp. 243–254 (IEEE, 2016).

Han, S., H. Mao and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding", arXiv preprint arXiv:1510.00149 (2015).

He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 770–778 (2016).

Heaton, J., N. Polson and J. H. Witte, "Deep learning for finance: deep portfolios", Applied Stochastic Models in Business and Industry **33**, 1, 3–12 (2017).

Hong, S., S. Kim, M. Joh and S.-k. Song, "Globenet: Convolutional neural networks for typhoon eye tracking from remote sensing imagery", arXiv preprint arXiv:1708.03417 (2017).

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

Hu, M., H. Li, Y. Chen, Q. Wu and G. S. Rose, "Bsb training scheme implementation on memristor-based circuit", in "Computational Intelligence for Security and Defense Applications (CISDA), 2013 IEEE Symposium on", pp. 80–87 (IEEE, 2013).

Huai, Y., "Spin-transfer torque mram (stt-mram): Challenges and prospects", AAPPS bulletin **18**, 6, 33–40 (2008).

Huval, B., T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue *et al.*, "An empirical evaluation of deep learning on highway driving", arXiv preprint arXiv:1504.01716 (2015).

Iandola, F., M. Moskewicz, S. Karayev, R. Girshick, T. Darrell and K. Keutzer, "Densenet: Implementing efficient convnet descriptor pyramids", arXiv preprint arXiv:1404.1869 (2014).

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", in "Proceedings of the 22nd ACM international conference on Multimedia", pp. 675–678 (ACM, 2014).

Jo, S. H., T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems", Nano letters **10**, 4, 1297–1301 (2010).

Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit", in "Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on", pp. 1–12 (IEEE, 2017).

Kadambi, P., A. Mohanty, H. Ren, J. Smith, K. McGuinnes, K. Holt, A. Furtwaengler, R. Slepetys, Z. Yang, J.-s. Seo *et al.*, "Towards a wearable cough detector based on neural networks", in "2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 2161–2165 (IEEE, 2018).

Kadetotad, D., Z. Xu, A. Mohanty, P.-Y. Chen, B. Lin, J. Ye, S. Vrudhula, S. Yu, Y. Cao and J.-s. Seo, "Neurophysics-inspired parallel architecture with resistive crosspoint array for dictionary learning", in "Biomedical Circuits and Systems Conference (BioCAS), 2014 IEEE", pp. 536–539 (IEEE, 2014).

Kadetotad, D., Z. Xu, A. Mohanty, P.-Y. Chen, B. Lin, J. Ye, S. B. Vrudhula, S. Yu, Y. Cao and J.-s. Seo, "Parallel architecture with resistive crosspoint array for dictionary learning acceleration.", IEEE J. Emerg. Sel. Topics Circuits Syst. **5**, 2, 194–204 (2015).

Kallenberg, M., K. Petersen, M. Nielsen, A. Y. Ng, P. Diao, C. Igel, C. M. Vachon, K. Holland, R. R. Winkel, N. Karssemeijer *et al.*, "Unsupervised deep learning applied to breast density segmentation and mammographic risk scoring", IEEE transactions on medical imaging **35**, 5, 1322–1331 (2016).

Karpathy, A., G. Toderici, S. Shetty, T. Leung, R. Sukthankar and L. Fei-Fei, "Large-scale video classification with convolutional neural networks", in "Proceedings of the IEEE conference on Computer Vision and Pattern Recognition", pp. 1725–1732 (2014).

Kehoe, B., S. Patil, P. Abbeel and K. Goldberg, "A survey of research on cloud robotics and automation.", IEEE Trans. Automation Science and Engineering **12**, 2, 398–409 (2015).

Kerem, E., S. Hirawat, S. Armoni, Y. Yaakov, D. Shoseyov, M. Cohen, M. Nissim-Rafinia, H. Blau, J. Rivlin, M. Aviram *et al.*, "Effectiveness of ptc124 treatment of cystic fibrosis caused by nonsense mutations: a prospective phase ii trial", The Lancet **372**, 9640, 719–727 (2008).

Kim, M., A. Mohanty, D. Kadetotad, N. Suda, L. Wei, P. Saseendran, X. He, Y. Cao and J.-s. Seo, "A real-time 17-scale object detection accelerator with adaptive 2000-stage classification in 65nm cmos", in "Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific", pp. 21–22 (IEEE, 2017).

Klein, G., Y. Kim, Y. Deng, J. Senellart and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation", arXiv preprint arXiv:1701.02810 (2017).

Krizhevsky, A. and G. Hinton, "Learning multiple layers of features from tiny images", Tech. rep., Citeseer (2009).

Krizhevsky, A., I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in "Advances in neural information processing systems", pp. 1097–1105 (2012).

Lai, S., L. Xu, K. Liu and J. Zhao, "Recurrent convolutional neural networks for text classification.", in "AAAI", vol. 333, pp. 2267–2273 (2015).

LeCun, Y., B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard and L. D. Jackel, "Handwritten digit recognition with a back-propagation network", in "Advances in neural information processing systems", pp. 396–404 (1990).

LeCun, Y., L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", Proceedings of the IEEE **86**, 11, 2278–2324 (1998).

LeCun, Y., C. Cortes and C. Burges, "Mnist handwritten digit database. at&t labs", (2010).

Lee, S. R., Y.-B. Kim, M. Chang, K. M. Kim, C. B. Lee, J. H. Hur, G.-S. Park, D. Lee, M.-J. Lee, C. J. Kim *et al.*, "Multi-level switching of triple-layered taox rram with excellent reliability for storage class memory", in "VLSI Technology (VLSIT), 2012 Symposium on", pp. 71–72 (IEEE, 2012).

Lemley, J., S. Bazrafkan and P. Corcoran, "Deep learning for consumer devices and services: Pushing the limits for machine learning, artificial intelligence, and computer vision.", IEEE Consumer Electronics Magazine **6**, 2, 48–56 (2017).

Levine, S., P. Pastor, A. Krizhevsky, J. Ibarz and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection", The International Journal of Robotics Research **37**, 4-5, 421–436 (2018).

Li, H., Z. Lin, X. Shen, J. Brandt and G. Hua, "A convolutional neural network cascade for face detection", in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition", pp. 5325–5334 (2015).

Liang, J., S. Yeh, S. S. Wong and H.-S. P. Wong, "Effect of wordline/bitline scaling on the performance, energy consumption, and reliability of cross-point memory array", ACM Journal on Emerging Technologies in Computing Systems (JETC) **9**, 1, 9 (2013).

Liu, B., H. Li, Y. Chen, X. Li, T. Huang, Q. Wu and M. Barnell, "Reduction and ir-drop compensations techniques for reliable neuromorphic computing systems", in "Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design", pp. 63–70 (IEEE Press, 2014).

Liu, B., H. Li, Y. Chen, X. Li, Q. Wu and T. Huang, "Vortex: variation-aware training for memristor x-bar", in "Proceedings of the 52nd Annual Design Automation Conference", p. 15 (ACM, 2015a).

Liu, J.-M., M. You, Z. Wang, G.-Z. Li, X. Xu and Z. Qiu, "Cough event classification by pretrained deep neural network", BMC medical informatics and decision making **15**, 4, S2 (2015b).

Ly, N., L. McCaig and C. W. Burt, "National hospital ambulatory medical care survey: 1999 outpatient department summary", Advance data from vital and health statistics , 321 (1999).

Ma, Y., Y. Cao, S. Vrudhula and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks", in "Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 45–54 (ACM, 2017).

Manyika, J., "A future that works: Ai, automation, employment, and productivity", (2017).

Marsden, P. A., J. A. Smith, A. A. Kelsall, E. Owen, J. R. Naylor, D. Webster, H. Sumner, U. Alam, K. McGuinness and A. A. Woodcock, "A comparison of objective and subjective measures of cough in asthma", Journal of Allergy and Clinical Immunology **122**, 5, 903–907 (2008).

Mathias, M., R. Benenson, M. Pedersoli and L. Van Gool, "Face detection without bells and whistles", in "European conference on computer vision", pp. 720–735 (Springer, 2014).

McGuinness, K., K. Holt, R. Dockry and J. Smith, "P159 validation of the vitalojak 24 hour ambulatory cough monitor", Thorax **67**, Suppl 2, A131–A131 (2012).

McGuinness, K., A. Morice, A. Woodcock and J. Smith, “The leicester cough monitor: a semi-automated, semi-validated cough detection system?”, European Respiratory Journal **32**, 2, 529–530 (2008).

Mohanty, A., X. Du, P.-Y. Chen, J.-s. Seo, S. Yu and Y. Cao, “Random sparse adaptation for accurate inference with inaccurate multi-level rram arrays”, in “Electron Devices Meeting (IEDM), 2017 IEEE International”, pp. 6–3 (IEEE, 2017).

Mohanty, A., N. Suda, M. Kim, S. Vrudhula, J.-s. Seo and Y. Cao, “High-performance face detection with cpu-fpga acceleration”, in “Circuits and Systems (ISCAS), 2016 IEEE International Symposium on”, pp. 117–120 (IEEE, 2016).

Nair, V. and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines”, in “Proceedings of the 27th international conference on machine learning (ICML-10)”, pp. 807–814 (2010).

Olshausen, B. A. and D. J. Field, “Emergence of simple-cell receptive field properties by learning a sparse code for natural images”, Nature **381**, 6583, 607 (1996).

Palossi, D., A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza and L. Benini, “Ultra low power deep-learning-powered autonomous nano drones”, arXiv preprint arXiv:1805.01831 (2018).

Rajendran, B., Y. Liu, J.-s. Seo, K. Gopalakrishnan, L. Chang, D. J. Friedman and M. B. Ritter, “Specifications of nanoscale devices and circuits for neuromorphic computational systems”, IEEE Transactions on Electron Devices **60**, 1, 246–253 (2013).

Rajpurkar, P., J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya *et al.*, “Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning”, arXiv preprint arXiv:1711.05225 (2017).

Raoux, S., G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, “Phase-change random access memory: A scalable technology”, IBM Journal of Research and Development **52**, 4.5, 465–479 (2008).

Rastegari, M., V. Ordonez, J. Redmon and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks”, in “European Conference on Computer Vision”, pp. 525–542 (Springer, 2016).

Seide, F., G. Li and D. Yu, “Conversational speech transcription using context-dependent deep neural networks”, in “Twelfth annual conference of the international speech communication association”, (2011).

Seo, J.-s., B. Lin, M. Kim, P.-Y. Chen, D. Kadetotad, Z. Xu, A. Mohanty, S. Vrudhula, S. Yu, J. Ye *et al.*, “On-chip sparse learning acceleration with cmos and resistive synaptic devices”, IEEE Trans. Nanotechnol **14**, 6, 969–979 (2015).

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search", nature **529**, 7587, 484 (2016).

Simonyan, K. and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", arXiv preprint arXiv:1409.1556 (2014).

Smith, J. and A. Woodcock, "New developments in the objective assessment of cough", Lung **186**, 1, 48–54 (2008).

Smith, J. A., H. L. Ashurst, S. Jack, A. A. Woodcock and J. E. Earis, "The description of cough sounds by healthcare professionals", Cough **2**, 1, 1 (2006).

Song, S., K. D. Miller and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity", Nature neuroscience **3**, 9, 919 (2000).

Sotelo, J., S. Mehri, K. Kumar, J. F. Santos, K. Kastner, A. Courville and Y. Bengio, "Char2wav: End-to-end speech synthesis", (2017).

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting", The Journal of Machine Learning Research **15**, 1, 1929–1958 (2014).

Suda, N., V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks", in "Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 16–25 (ACM, 2016).

Swarnkar, V., U. R. Abeyratne, Y. Amrulloh, C. Hukins, R. Triasih and A. Setyati, "Neural network based algorithm for automatic identification of cough sounds", in "2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)", pp. 1764–1767 (IEEE, 2013).

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going deeper with convolutions", in "Proceedings of the IEEE conference on computer vision and pattern recognition", pp. 1–9 (2015).

Tao, Y., X. Gao, K. Hsu, S. Sorooshian and A. Ihler, "A deep neural network modeling framework to reduce bias in satellite precipitation products", Journal of Hydrometeorology **17**, 3, 931–945 (2016).

Theis, T. N. and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology", Computing in Science & Engineering **19**, 2, 41–50 (2017).

Tosic, I. and P. Frossard, "Dictionary learning", IEEE Signal Processing Magazine **28**, 2, 27–38 (2011).

Tracey, B. H., G. Comina, S. Larson, M. Bravard, J. W. López and R. H. Gilman, "Cough detection algorithm for monitoring patient recovery from pulmonary tuberculosis", in "Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE", pp. 6017–6020 (IEEE, 2011).

Tseng, K.-L., Y.-L. Lin, W. Hsu and C.-Y. Huang, “Joint sequence learning and cross-modality convolution for 3d biomedical segmentation”, in “Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on”, pp. 3739–3746 (IEEE, 2017).

Viola, P. and M. Jones, “Rapid object detection using a boosted cascade of simple features”, in “Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on”, vol. 1, pp. I–I (IEEE, 2001).

Wang, Z., “Cmos adjustable schmitt triggers”, IEEE Transactions on instrumentation and Measurement **40**, 3, 601–605 (1991).

Whaley, R. C. and J. J. Dongarra, “Automatically tuned linear algebra software”, in “Supercomputing, 1998. SC98. IEEE/ACM Conference on”, pp. 38–38 (IEEE, 1998).

Wong, H.-S. P., H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen and M.-J. Tsai, “Metal–oxide rram”, Proceedings of the IEEE **100**, 6, 1951–1970 (2012).

Xia, L., T. Tang, W. Huangfu, M. Cheng, X. Yin, B. Li, Y. Wang and H. Yang, “Switched by input: power efficient structure for rram-based convolutional neural network”, in “Proceedings of the 53rd Annual Design Automation Conference”, p. 125 (ACM, 2016).

Xu, Z., A. Mohanty, P.-Y. Chen, D. Kadetotad, B. Lin, J. Ye, S. Vrudhula, S. Yu, J.-s. Seo and Y. Cao, “Parallel programming of resistive cross-point array for synaptic plasticity”, Procedia Computer Science **41**, 126–133 (2014).

Xu, Z., S. Skorheim, M. Tu, V. Berisha, S. Yu, J.-s. Seo, M. Bazhenov and Y. Cao, “Improving efficiency in sparse learning with the feedforward inhibitory motif”, Neurocomputing **267**, 141–151 (2017).

You, J., X. Li, M. Low, D. Lobell and S. Ermon, “Deep gaussian process for crop yield prediction based on remote sensing data.”, in “AAAI”, pp. 4559–4566 (2017).

Young, E. C. and J. A. Smith, “Quality of life in patients with chronic cough”, Therapeutic advances in respiratory disease **4**, 1, 49–55 (2010).

Young, T., D. Hazarika, S. Poria and E. Cambria, “Recent trends in deep learning based natural language processing”, ieee Computational intelligenCe magazine **13**, 3, 55–75 (2018).

Yu, S., B. Gao, Z. Fang, H. Yu, J. Kang and H.-S. P. Wong, “A low energy oxide-based electronic synaptic device for neuromorphic visual systems with tolerance to device variation”, Advanced Materials **25**, 12, 1774–1779 (2013).

Zhang, C., P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks", in "Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 161–170 (ACM, 2015).

Zhu, X. and D. Ramanan, "Face detection, pose estimation, and landmark localization in the wild", in "Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on", pp. 2879–2886 (IEEE, 2012).