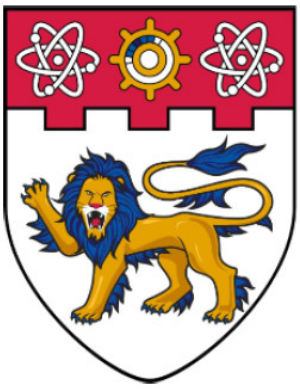


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

COMPACT AND FAST MACHINE LEARNING ACCELERATOR FOR IOT DEVICES

HUANG HANTAO
SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING
2018

COMPACT AND FAST MACHINE LEARNING ACCELERATOR FOR IOT DEVICES

HUANG HANTAO

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2018

Acknowledgement

Firstly, I would like to express my sincere appreciation to my advisors, Professor Goh Wang Ling and Professor Yu Hao. My special thanks goes to Prof. Goh, who has supported and guided me for graduation. I also greatly appreciate Prof. Yu's extraordinary foresight, which constantly drives me to keep eyes on real-life problems and has directed my research to bridge theory with practical applications. He has set a great example of researcher with enthusiasm, critical thinking and hard working on research. I am also indebted to him for his encouraging support for exchanging to Georgia Institute of Technology and recommending me to my industrial jobs.

Secondly, I am very grateful towards the confirmation exercise examiners Prof. Jong Ching Chuen, Prof. Jiang Xudong and Prof. Anupam Chattopadhyay for accessing my research and providing me guidance. My special thanks goes to Prof. Ren Fengbo, who has taught me on FPGA design and providing me many suggestions on algorithm optimization. I would also like to thank the staff of graduate pro gramme office for their assistance in administrative matters for my graduate studies.

Thirdly, I want to express my deepest appreciation to thesis advisor committee (TAC) members, Prof. Anupam Chattopadhyay and Prof. Goh Wang Ling, who have gave me many guidance during my research and helped me prepare my thesis. The special thanks goes to VIRTUS staff, David Robert Neubronner, who helps me set up various software and systems. Moreover, I would like to express my thanks to my wonderful colleagues, Xu Hang, Liu Zichuan, Ni Leibin, Cai Yuehua, Xu Dongjun, Sai Manoj P.D , Huang Xiwei, Wang Xiaolong, Chen Shuai, Yang Chang and many other people who accompanied me to work together for so many important years.

The research presented in the thesis is funded by JIP-Mediatek Scholarship. The special thanks goes to Mediatek Singapore staff, Dr. Hao Zhigang, who has guided me on the research direction and discussed my potential future jobs.

Most importantly, I would like to thank my parents and my girl friend for their endless love and support, which have always been giving me the strength and courage to overcome difficulties.

Abstract

The Internet of things (IoT) is the networked interconnection of every object to provide intelligent service and improve economy benefit. The potential of IoT and its ubiquitous computation reality are staggering, but limited by many technical challenges. One challenge is to have a real-time response to the dynamic ambient change. Machine learning accelerator on IoT edge devices is one potential solution since a centralized system suffers long latency of processing in the back end. However, IoT edge devices are resource-constrained and machine learning algorithms are computational intensive. Therefore, optimized machine learning algorithms, such as compact machine learning for less memory usage on IoT devices, is greatly needed. In this thesis, we explore the development of fast and compact machine learning accelerators by developing least-squares solver, tensor-solver and distributed-solver. Moreover, applications such as energy management system using such machine learning solver on IoT devices are also investigated.

From the fast machine learning perspective, the target is to perform fast learning on the neural network. This thesis proposes a least-squares-solver for single hidden layer neural network. Furthermore, this thesis explores the CMOS FPGA based hardware accelerator and RRAM based hardware accelerator. A 3D multi-layer CMOS-RRAM accelerator architecture for incremental machine learning is proposed. By utilizing an incremental least-squares solver, the whole training process can be mapped on the 3D multi-layer CMOS-RRAM accelerator with significant speed-up and energy-efficiency improvement. Experimental results using the benchmark CIFAR-10 show that the proposed accelerator has 2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC hardware implementation; and 14.94× speed-up, 447.17× energy-saving and around 164.38× area-saving compared to CPU software implementation. Compared to GPU implementation, our work shows 3.07× speed-up and 162.86× energy-saving. In addition, a CMOS based FPGA realization of neural network with square-root-free Cholesky factorization is also investigated for training and inference. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable accuracy with an average speed-up of 4.56× and 89.05×,

when compared to x86 CPU and ARM CPU for load forecasting in the energy management system.

From the compact machine learning perspective, this thesis proposes a tensor-solver for deep neural network compression with consideration of the accuracy. A layer-wise training of tensorized neural network (TNN) has been proposed to formulate multilayer neural network such that the weight matrix can be significantly compressed during training. By reshaping the multilayer neural network weight matrix into a high dimensional tensor with a low-rank approximation, significant network compression can be achieved with maintained accuracy. For MNIST benchmark, TNN shows 64× compression rate without accuracy drop. For CIFAR-10 benchmark, TNN shows that 21.57× compression rate is achieved for fully-connected layers with 2.2% accuracy drop. In addition, a highly-parallel yet energy-efficient machine learning accelerator has been proposed for such tensorized neural network. Moreover,simulation results using the benchmark MNIST show that the proposed CMOS-RRAM accelerator has 1.283× speed-up, 4.276× energy-saving and 9.339× area-saving compared to 3D CMOS-ASIC implementation; and 6.37× speed-up and 2612× energy-saving compared to 2D CPU implementation.

From large scaled IoT networks perspective, this thesis proposes a distributed-solver on IoT devices. Furthermore, this thesis proposes a distributed neural network and sequential learning on the smart gateways for indoor positioning, energy management and IoT network security. For indoor positioning system, experimental results show that the proposed algorithm can achieve 50× and 38× timing speed-up during inference and training respectively with comparable positioning accuracy, when compared to traditional support vector machine (SVM) method. For energy management system, experiment results on real-life datasets have shown that the accuracy of the proposed energy prediction can be 14.83% improvement comparing to SVM method. Moreover, the peak load from main electricity power-grid is reduced by 15.20% with 51.94% . For network intrusion detection of IoT systems, experimental results on a single FPGA achieve a bandwidth of 409.6 Gbps with 4.5× and 77.4× speed-up compared to general CPU and embedded CPU. Our FPGA accelerator provides a low-power and low-latency intrusion detection performance for the IoT network security.

In conclusion, this thesis investigates the compact and fast machine learning accelerator on IoT devices, which is desirable to build a real IoT system towards future smart home, smart building, smart community and further a smart city.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Internet of Things (IoT)

The term ”Internet of Things” refers to a networked infrastructure, where each object is connected with identity and intelligence [3]. The IoT infrastructure makes objects remotely connected and controlled. Moreover, intelligent IoT devices can understand the physical environment and thereby perform smart actions to optimize daily benefits such as improving resource efficiency. For example, the deployment of IoT devices for smart buildings and homes will perform energy saving with a high level of comforts. However, collecting personal daily information and uploading them to the cloud may bear the risk of sensitive information leakage. Furthermore, the large volume of data generated by IoT devices poses a great challenge on current cloud based computation platform. For example, a running car will generate one Gigabyte data every second and it requires real-time data processing for vehicle to make correct decisions [4]. The current network is not be able to perform such large volume of data communication in a reliable and real-time fashion [5]. Considering these challenges, an edge device based computation in IoT networks becomes more preferred. The motivation of edge device computation can be summarized from two manifold. Firstly, it preserves information privacy. It can analyze the sensitive information locally to perform the task or pre-process the sensitive data before sending to the cloud. Secondly, computation on edge devices can reduce the latency. Edge computing application can implement machine learning algorithm directly on IoT devices to perform the task, which can reduce the latency and become robust to connectivity issues.

Fig. 1.1 shows the comparisons of IoT networked devices. Edge devices are mainly resource-constrained devices with limited memory. To run machine learning algorithms on such devices, the co-design of computing architecture and algorithm for performance

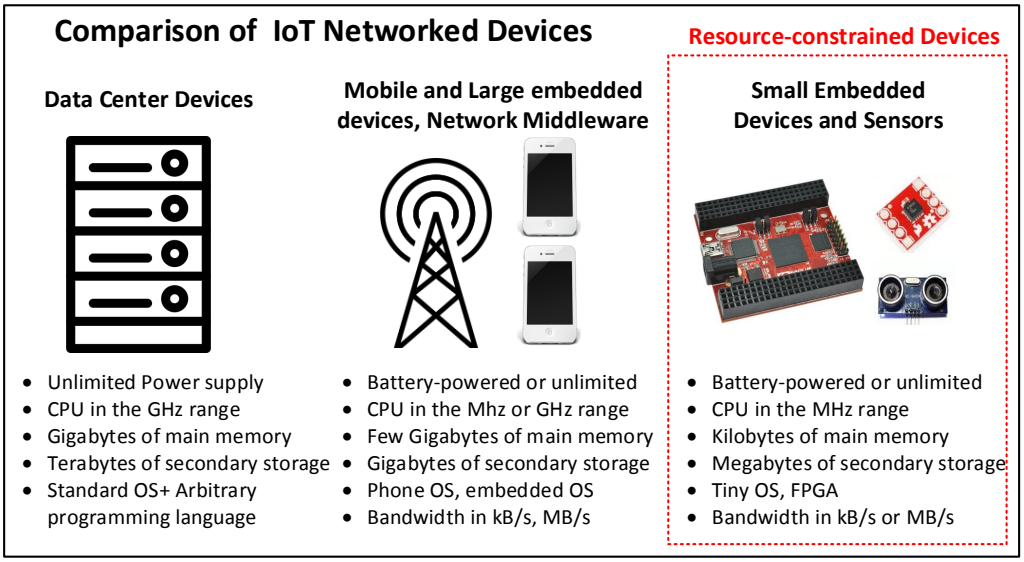


Figure 1.1: Comparison of computing environments and device types

optimization is greatly required. Therefore, in the following section, we will discuss the machine learning accelerator using edge IoT devices.

1.2 Machine Learning Accelerator

The trend of using deeper neural network for machine learning has introduced a grand challenge of high-throughput yet energy-efficient hardware accelerators [6, 7]. Co-design of neural network compression algorithm as well as computing architecture is required to tackle the complexity [8].

From the neural network compression algorithm perspective, connection pruning, weights shearing, weights quantization [9, 10] are widely applied to compress the neural network model. [11] further adopted low-rank approximation directly to the weight matrix after training. However, such directly approximated computing can simply reduce complexity but cannot maintain the accuracy, especially when simplification is performed to the network obtained after the training without fine-tuning. In contrast, many recent works [12] have found that the accuracy can be maintained when some constraints such as sparsity are applied during the training.

From the computing hardware architecture perspective, due to the large size of training data and the limited parallel processing capability of general purpose processors, the training process of machine learning can take up to a few weeks running on CPU clusters, making timely assessment of model performance impossible. This has forced the engineers to take a parallelization of experiments approach in model design, which

demands a huge amount of computing resources with poor energy efficiency. Recently, graphic processing units (GPUs) have been widely adopted for accelerating deep neural network (DNN) due to their large memory bandwidth and high parallelism of computing resources. However, the undesirably high power consumption of high-end GPUs presents significant challenges to IoT systems. The low power CMOS-based accelerator becomes a potential solution. Considering the dynamic change of IoT environments, a reconfigurable FPGA becomes more preferred for different application requirements. In addition, the recent resistive random access memory (RRAM) devices [13, 14] have shown great potential for an energy-efficient in-memory computation of neural networks. It can be exploited as both storage and computation elements with minimized leakage power due to its non-volatility. The latest works in [13, 14] show that the 3D CMOS-RRAM integration can further support more parallelism with higher I/O bandwidth in acceleration. Therefore, in this thesis, we will investigate the fast machine learning accelerator on both CMOS and RRAM based computing systems.

1.3 Contribution of This Work

Although machine learning algorithms implemented directly on edge devices seem to be a promising solution for IoT applications, there are still challenges to be tackled on the resource-constrained IoT devices. In this thesis, we explore the development of fast and compact machine learning accelerators by developing least-squares solver, tensor-solver and distributed-solver. Moreover, applications of such machine learning solver on IoT devices are also studied. The main contribution of this thesis can be summarized as follows.

For the fast machine learning accelerator, the target is to perform fast learning on the neural network. This thesis proposes a least-squares-solver for single hidden layer neural network. Furthermore, this thesis explores the CMOS FPGA based hardware accelerator and RRAM based hardware accelerator. Particularly, this thesis has summarized the following works:

- A fast machine learning accelerator on FPGA for real-time data analytics in smart micro-grid of buildings is proposed. An incremental and square-root-free Cholesky factorization algorithm is introduced with FPGA realization for training acceleration when analyzing the real-time sensed data. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable load forecasting accuracy with an average speed-up of 4.56× and 89.05×, when compared to x86 CPU and ARM CPU. Moreover , 450.2×, 261.9× and 98.92× energy saving

can be achieved comparing to x86 CPU, ARM CPU and GPU respectively.

- A 3D multi-layer CMOS-RRAM accelerator architecture for fast machine learning is proposed. By utilizing an incremental least-squares solver, the whole training process can be mapped to the 3D multi-layer CMOS-RRAM accelerator with significant speed-up and energy-efficiency improvement. Experimental results using the benchmark CIFAR-10 show that the proposed accelerator has 2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC hardware implementation; and 14.94× speed-up, 447.17× energy-saving and around 164.38× area-saving compared to CPU software implementation. Compared to GPU implementation, our work shows 3.07× speed-up and 162.86× energy-saving.

For the compact machine learning accelerator, this thesis investigates a tensor-solver for deep neural network with neural network compression. The previous solution is mainly based on bit-width truncation and hashing trick, which cannot optimize the both accuracy and model size. On the other hand, representing each weight as a high dimensional tensor and then performing tensor-train decomposition can effectively reduce the size of weight matrix (number of parameters). Particularly, this thesis has summarized the following works:

- A layer-wise training of tensorized neural network (TNN) has been proposed to formulate multilayer neural network such that the weight matrix can be significantly compressed during training. By reshaping the multilayer neural network weight matrix into a high dimensional tensor with a low-rank approximation, significant network compression can be achieved with maintained accuracy. A layer-wise training is developed by a modified alternating least-squares (MALS) method without backward propagation (BP). TNN can provide state-of-the-art results on various benchmarks with significant compression. For MNIST benchmark, TNN shows 64× compression rate without accuracy drop. For CIFAR-10 benchmark, TNN shows that compression of 21.57× can be achieved for fully-connected layers with 2.2% accuracy drop.

- A highly-parallel yet energy-efficient machine learning accelerator has been proposed for tensorized neural network. Highly parallel matrix-vector multiplication can be performed with low power in the proposed 3D multi-layer CMOS-RRAM accelerator. The adoption of tensorization can significantly compress the weight matrix of neural network using much fewer parameters. Simulation results using

the benchmark MNIST show that the proposed accelerator has 1.283× speed-up, 4.276× energy-saving and 9.339× area-saving compared to 3D CMOS-ASIC implementation; and 6.37× speed-up and 2612× energy-saving compared to 2D CPU implementation. In addition, 14.85× model compression can be achieved by tensorization with acceptable accuracy loss.

For the large scaled IoT network, this thesis proposes a distributed-solver on IoT devices. Furthermore, this thesis proposes a distributed neural network and sequential learning on the smart gateways for indoor positioning, energy management and IoT network security. Particularly, this thesis has summarized the following works:

- A computationally efficient data analytics by distributed nerual network (DNN) based machine learning is proposed with application for indoor positioning. It is based on one incremental least-squares solver for learning collected WiFi-data at each gateway and is further fused for all gateways in the network to determine the location. Experimental results show that with multiple distributed gateways running in parallel, the proposed algorithm can achieve 50× and 38× speed-up during inference and training respectively with comparable positioning accuracy, when compared to traditional support vector machine (SVM) method.

- A distributed and networked machine learning platform on smart gateways is proposed for energy management system with consideration of both occupant profile and energy profile. It can analyze occupants motion, provide short-term energy forecast and allocate renewable energy resource. Firstly, occupant profile is captured by real-time indoor positioning system with Wi-Fi data analytics; and the energy profile is extracted by real-time meter system with electricity load data analytics. Then, the 24-hour occupant profile and energy profile are fused with prediction using an on-line distributed machine learning with real-time data update. Based on the forecast occupant motion profile and energy consumption profile, solar energy is allocated on the additional electricity power-grid in order to reduce peak demand on the main electricity power-grid. The whole management flow can be operated on the distributed smart gateway network with limited computation resource but with a supported general machine learning engine. Experiment results on real-life datasets have shown that the accuracy of the proposed energy prediction can be 14.83% improvement comparing to SVM method. Moreover, the peak load from main electricity power-grid is reduced by 15.20% with 51.94% energy cost saving.

- An online sequential machine learning hardware accelerator is proposed to perform real-time network intrusion detection for IoT networks. A neural network based learning algorithm is developed with an incremental least-squares-solver realized on hardware. A fast and low-power IoT NIDS can be achieved using hardware accelerator with sequential learning to adapt to new threats. Furthermore, a single hidden layer feedforward neural network based learning algorithm is developed with an incremental least-squares solver realized on hardware. Experimental results on a single FPGA achieve a bandwidth of 409.6 Gbps with 4.5× and 77.4× speed-up compared to general CPU and embedded CPU. Our FPGA accelerator provides a low-power and low-latency intrusion detection performance for the IoT network security.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents an overview of IoT systems and machine learning algorithms. Chapter 3 introduces a least-squares-solver for single hidden layer neural network. The training process is optimized and mapped on both CMOS FPGA and RRAM devices. A detailed tensorized neural network with hardware implementation is presented in Chapter 4. With the adoption of tensor-train data format, significant network compression can be achieved. In Chapter 5, a distributed neural network with online sequential learning is studied. The application of such distributed neural network is discussed in the smart building environment. With such common machine learning engine, energy management, indoor positioning and network security can be performed. Conclusions are drawn in Chapter 6 with suggestions for future works.

Chapter 2

Fundamentals and Literature Review

2.1 Edge Computing on IoT Devices

By 2020, there will be over 20 billion devices connected to IoT devices [15]. As the number of IoT devices is increasing tremendously, one grand challenge is how to perform timely data collection and analysis through the IoT devices. Traditionally, IoT data is collected locally and sent to servers or data centers for data analytics, so called cloud computing. This is a centralized approach but suffers from the long data communication latency as well as large power consumption. Moreover, since the IoT data volume is increasing exponentially, data center becomes non-scalable. As such, there is an increasing demand to develop a new computing platform to alleviate the heavy computation and communication load on data center, so called edge computing, which can provide a timely decision-making.

Based on the computing capability of smart sensors and smart gateways, edge computing becomes a potential solution to perform real-time data analytics. It refers to data analytics at the edge of networks, close to the source of sensed data at IoT devices. The data can be analyzed by ARM cores, FPGAs or ASICs on the edge IoT devices or networks, comparing to the one analyzed by CPUs and GPUs in the data center. With the edge computing, the sensed data is pre-processed or even analyzed locally. Fig. 2.1

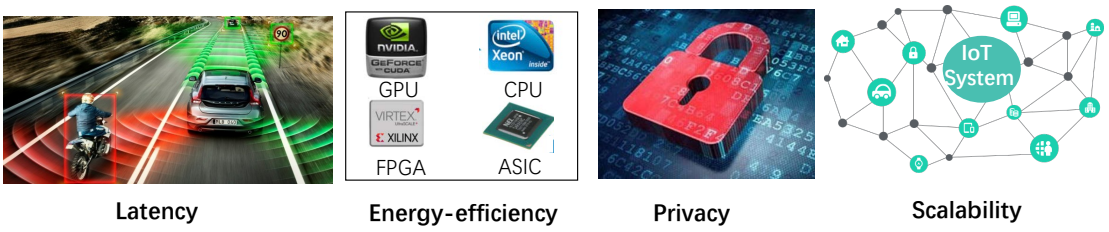


Figure 2.1: Motivation to use edge computing in IoT systems

shows the benefit of edge computing on IoT devices. Obviously, latency and power will be reduced by edge computing on IoT devices. Moreover, data privacy can be protected by performing data analytics locally, because only pre-processed data is sent to the data servers to avoid sensitive information leakage. More importantly, the scalability of IoT devices based edge computing is also improved since the computation can be performed in a distributed fashion.

However, the IoT devices have limited computational resource, it is still unknown or unsatisfied on how to develop data analytics on the edges IoT devices such as machine learning algorithms. The traditional data analytics by machine learning requires intensive computation. In this thesis, we will show three different machine-learning solvers: least-squares-solver, tensor-solver and distributed-solver, which can be adopted on the edge IoT devices. The least-squares-solver is designed for small-sized neural network training problems. The tensor-solver is designed to compress the deep neural networks such that modern multi-layer neural networks can be mapped on IoT devices. The distributed-solver is further proposed to improve the performance for large-scaled IoT networks.

In the following of this chapter, we will use IoT-based smart buildings as one example to illustrate how to perform edge computing on IoT devices with applications such as: indoor positioning, energy management and network intrusion detection. Moreover, we will also discussed the basics of the machine learning algorithms, distributed machine learning, machine learning accelerators and machine learning model optimization.

2.2 IoT based Smart Buildings

Buildings are complex electrical and mechanical systems. Deploying sensors into building can monitor the building operation such as the utility consumption. Integrating machine learning into the building management system can optimize the control of heating, cooling and lighting. Moreover, with machine learning techniques, system can not only understand the environment but also make optimal decisions to assist living and perform energy saving. Fig. 2.2 shows a typical IoT based smart building. A data hub will collect various data from sensors and send them to smart gateways. Machine learning algorithms will directly run on the smart gateways locally to perform real-time responses and protect privacy. Smart services such as energy management and security will be provided through GUI. Such IoT based smart buildings can improve energy-efficiency as well as assist occupant living. This thesis mainly focuses on three areas of IoT based service: indoor positioning, energy management and security. They are elaborated in details in the following sections.

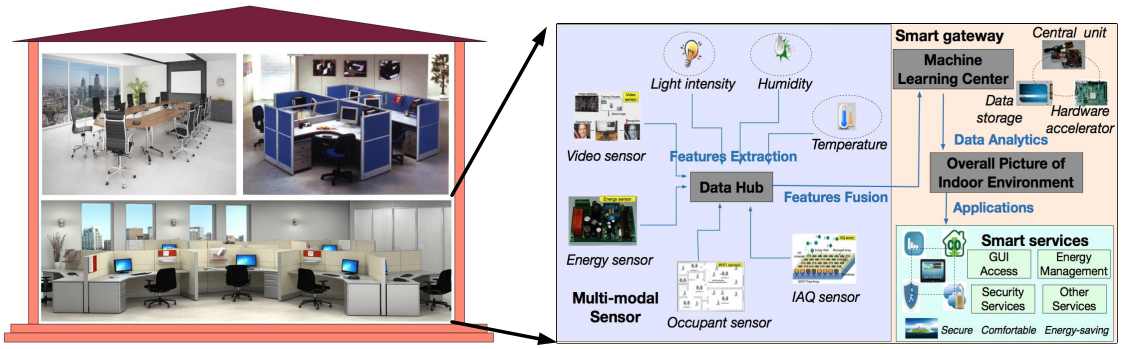


Figure 2.2: IoT based smart buildings with various sensor and smart gateways to provide smart services

2.2.1 IoT based Indoor Positioning System

GPS provides excellent outdoor services, but due to the lack of Line of Sight (LoS) transmissions between the satellites and the receivers, it is not capable of providing positioning services in indoor environment [16]. Developing a reliable and precise indoor positioning system (IPS) has been extensively researched as a compensation for GPS services in indoor environment. Wi-Fi based indoor positioning is becoming very popular these days due to its low cost, good noise immunity and low set-up complexity [17, 18]. Many WiFi-data based positioning systems have been developed recently based on received signal strength indicator (RSSI) [19]. As the RSSI parameter can show large dynamic change under environmental change (such as obstacles) [20, 21, 22], the traditional machine-learning based WiFi data analytic algorithms can not adapt to the environment change because of the large latency. This is mainly due to the centralized computational system and the high training complexity [23], which will introduce large latency and also cannot be adopted on the sensor network directly. Therefore, we propose to develop a distributed indoor positioning algorithm targeting to computational resource limited devices.

Existing positioning systems can be classified into symbolic and geometric models. In a symbolic model, all objects are represented as symbols and referred by names or labels; in a geometric model, the physical space is represented as the Euclidean space and objects are described by the set of coordinates in the Euclidean space. The coordinate system is not really suitable for indoor environment since each indoor environment has its special layout to describe its position [24]. Therefore, we adopt a symbolic model and a classification algorithm is required.

Machine learning algorithms are known to tackle noisy data and widely used when the correlation between input and output is unclear [25]. Machine learning based indoor

positioning refers to algorithms that first collect features (fingerprints) of a scene and then estimate the location by matching the online measurements with priori collected features. Therefore, for such algorithms, there are two stages: training stage and inference stage. During the training stage, RSSI of the radio is measured at predefined points in the environment. Radio feature such as signal strength is extracted and collected offline with a relationship model (called radio map) between distance and signal strength, which is described by a (distributed) neural network. During the inference stage, the received signal will be mapped to the most correlated features in the radio map and hence to estimate or calculate the location. For indoor location, during the inference stage, the number of layers for the neural network should be small for neural network processing to achieve real-time position tracking.

2.2.2 IoT based Energy Management System

According to the estimation by Harvey in 2010, the future global power requirement will reach 10 tera-watt scale [26]. Among various energy consumers, it has also been reported that over 70% electricity is consumed by more than 79 million residential buildings and 5 million commercial buildings in US [27]. As a result, an increasing demand is observed to design the automatic energy management system (EMS) for buildings, which relies heavily on machine learning techniques for energy resource scheduling by assessing existing electricity from external power-grid, commissioning new renewable solar energy, evaluating service contract options, and optimizing EMS operations [28]. Although the traditional centralized and static EMS has been successfully utilized to provide stable energy supply, new challenges emerge for building one scalable, highly efficient yet stochastically optimized EMS in the smart grid era.

From the energy supplier side, one of the most important feature of future smart grid or smart building is the deployment of renewable energy such as solar and wind energy. For example, in addition to the traditional bulk power generators, many small but distributed photovoltaic (PV) panels or wind turbines have been equipped to supply clean energies to modern buildings [29, 30]. From the energy user side, modern buildings typically accommodate a hybrid of hundreds of rooms, and each room has its unique energy usage characteristic. As such, the system complexity has grown rapidly due to the increased number of hybrid energy suppliers and users, which raises higher demand on the scalability of modern smart building EMS to achieve real time control.

As shown in Fig. 2.3, load profiles of residential rooms and commercial rooms show great variabilities. Even within the same type of rooms, the load profiles tend to differ from each other. Therefore, the load profile is not only affected by external factors

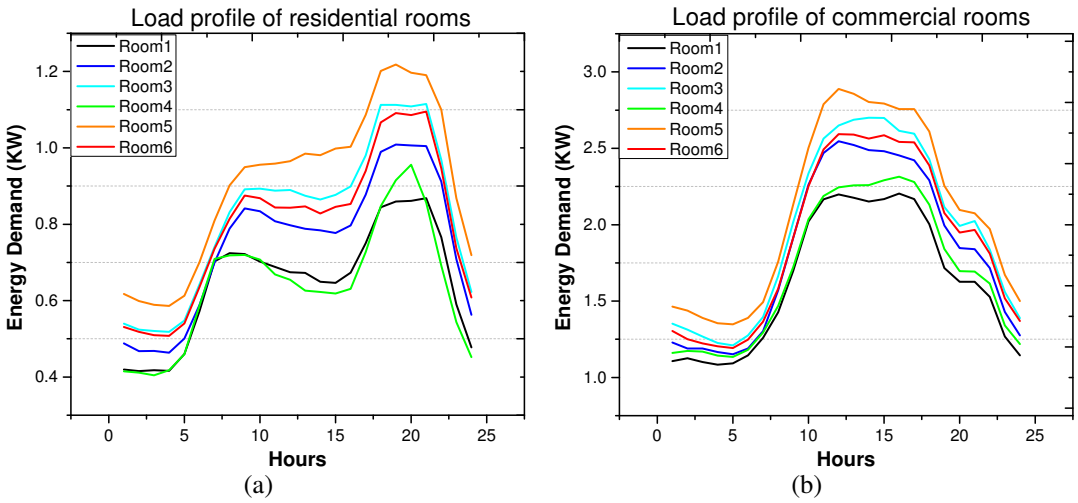


Figure 2.3: Workload profiles (summer) for (a) residential rooms; (b) commercial rooms.

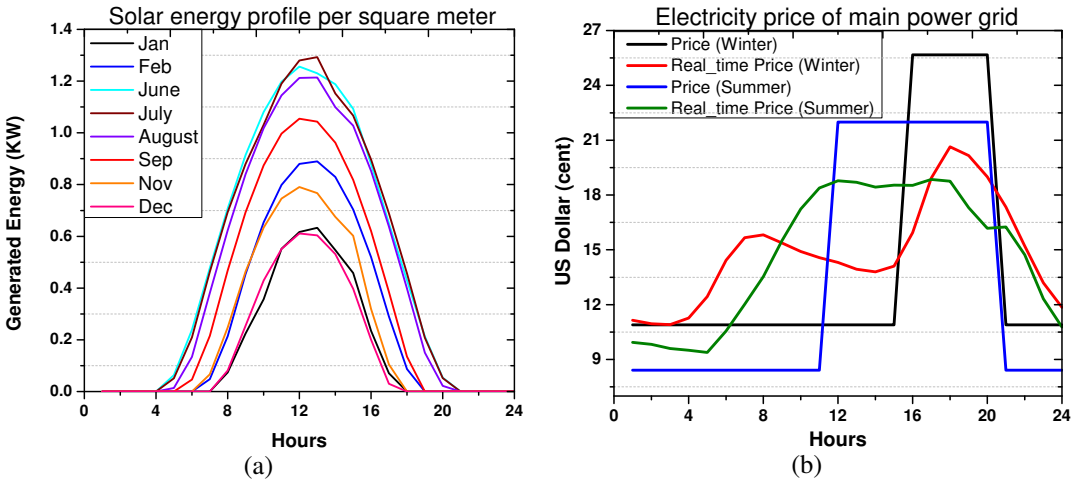


Figure 2.4: (a) Solar energy profile; (b) electricity price of main power grid.

but also occupants behaviors. Moreover, the generation of renewable solar energy is mismatch from the price of the electricity as shown in Fig. 2.4. Therefore, an accurate short-term load forecasting to make use of renewable energy is crucial to reduce the electricity cost. However, previous works focus on one or some parts of influential factors (e.g. season factors [31], cooling/heating factors [32] or environmental factors [33]) without consideration of occupant behaviors. The occupant behavior is one of the most influential factors affecting the energy load demand [1]. Understanding occupant behaviors can provide more accurate load forecasting and better energy saving without sacrificing comfort levels. As such, the building energy management system with

[1] Obviously, the occupant refers to the end users of buildings.

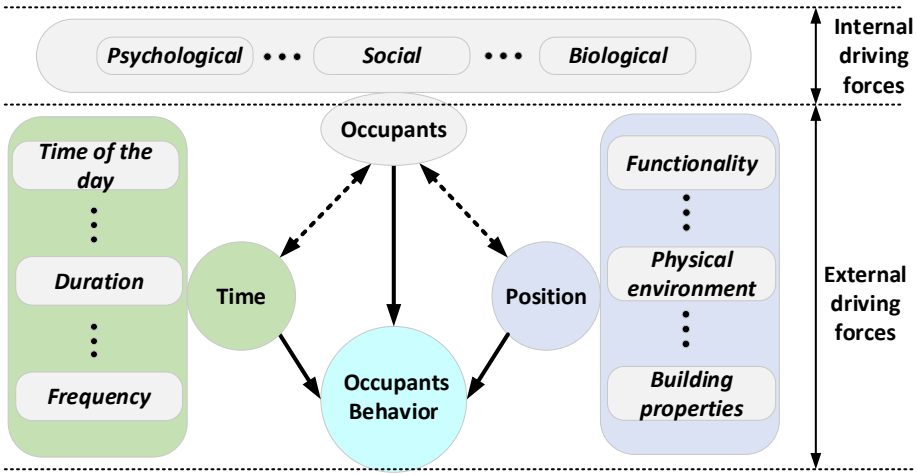


Figure 2.5: Driving forces of energy-related occupant behaviors

consideration of occupant behaviors is greatly required.

Previous works considering occupant behaviors are mainly based on statistical analysis. [34, 35] proposed to use hidden Markov chain to model occupant behaviors to perform energy management. However, such static modeling can not adapt to the change of environment as well as occupant behaviors. Recently, the advancement of IoT hardware enables the real-time data collection and analytics of occupant behaviors [36]. This can be done by analyzing occupant position, environmental factors and other external forces [37] as shown in Fig. 2.5. However, IoT hardwares are computational resource limited, which cannot perform intensive computation. Therefore, we propose a distributed machine learning on IoT hardware to perform occupant behavior analytics and then further to support smart-grid energy optimization.

2.2.3 IoT based Network Intrusion Detection System

Intrusion detection system (IDS) is designed to identify security violation in a computing system such as embedded systems [38]. Among various techniques used for intrusion detection, signature-based detection and anomaly-based detection are the most widely used [39]. Signature-based detection detects an intrusion by comparing events against known signatures for the intrusions whereas anomaly-based detection creates a behavior model of normal behaviors and detect the deviation from the normal behaviors. Although signature-based detection such as Snort [40] has very low false-alarm rates, this method suffers from the failure of detecting many attack variants [38]. Moreover, as new types of attack signatures increase, it requires frequent signature update and becomes impractical to store all the signatures, especially for IoT embedded systems. Instead, anomaly-based detection can detect unknown types of attacks and become one potential

solution for IoT network intrusion detection system (NIDS).

Traditionally, network intrusion detection is mainly performed by software based solutions on general purpose hardware. However, a real-time IoT network intrusion detection with low-power requirement is hard to achieve through general purpose hardware. A custom-tailored hardware such as embedded FPGA can achieve better energy-efficiency with higher detection speed. Previous works on hardware based intrusion detection are mainly based on machine learning pre-trained models [38, 41]. However, it suffers from two limitations. Firstly, such pre-trained model is relatively large and may not be suitable to map on resource limited IoT devices with real-time intrusion detection requirement [42]. Secondly, the update of detection engine requires reprogramming the entire FPGA chip with new training data [38], which reduces the performance of IoT services. Therefore, in this thesis, we propose an online sequential neural network learning accelerator for IoT NIDS. It can perform sequential learning for new identified attacks with fast and energy-efficient performance.

2.3 Machine Learning

2.3.1 Machine Learning Basics

The problem of finding patterns from sampled data has been extensively studied by research communities. Detecting patterns and recognizing them will help understand data and even generate new knowledge. To achieve these purposes, machine learning algorithms are proposed to perform automatic data processing to achieve pre-defined objectives. Machine learning algorithms are further divided into three classes according to the level of information provided [43], which are supervised learning, unsupervised learning and semi-supervised learning.

Supervised learning problem is to learn a function on all possible trajectories of the data based on the finite subset of the observed data, which is called the training set. Such training set is also associated with additional information to indicate their target values. The learning algorithm defines a function, which maps the training data to the target values. If the amount of the target values are finite, such problem is defined as classification problems. However if the amount of the target values are infinites, this problem is defined as regression problems.

Unsupervised learning is a learning algorithm to draw inference from datasets without labeled responses. One of the common methods is the clustering technique, which is used to perform data analysis to find hidden patterns from a group of data. One major

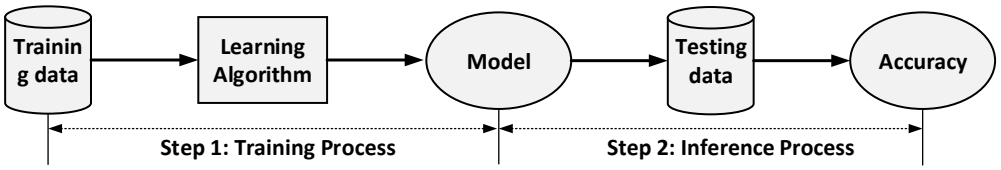


Figure 2.6: Supervised learning process:training and inference

challenge in the unsupervised learning is how to define the similarity between two features or input data. Another challenge is that different algorithms may lead to different results, which requires experts to analyze them.

Semi-supervised learning shares the same goal as the supervised learning [43]. However, now the designer has some unknown patterns and known patterns, which we usually call the former ones as unlabeled data and the latter as labeled data. Semi-supervised learning is designed when the designer has limited access to the labeled data.

In this thesis, we will focus on supervised learning. The supervised learning process as shown in Fig. 2.6 includes two processes: learning and inference [2] . Learning process is to generate a model based on the training data. Inference process is to use unseen data to assess the model accuracy. This model works under the assumption that the distribution of training samples is identical to the distribution of inference examples. This assumption requires that the training samples must be sufficiently representative of the inference data.

There are many supervised machine learning algorithms, such as decision trees, naive Bayesian, support vector machine (SVM) and neural networks. Decision trees such as C4.5 [44] is one of the most widely used techniques and can be transformed into rules based classification techniques. However, this method suffers from overfitting. Naive bayesian is to compute the maximum posteriori probability for the given inference sample. Such method is efficient and easy to implement. However, the assumption that each class is conditionally independent can be seriously violated. This results in significant loss of accuracy. Support vector machine (SVM) is invented by V. Vapnik [45]. SVM has a rigorous theoretical foundation, which finds a separating hyperplane with the largest margin to separate two classes of data, positive and negative. It works very well. However, it is mainly designed for binary classification.

The state-of-the-art machine learning algorithm is the neural network algorithm. A neural network consists of a pool of simple processing units, which receive input data from neighbors or previous layers and use the input to compute the output for the next layer. The neural network weight is adjusted during the training process to minimize

[2]Some literatures may indicate the inference process as the testing process. In this thesis, testing and inference are interchangeable and testing data refers to the data used for inference.

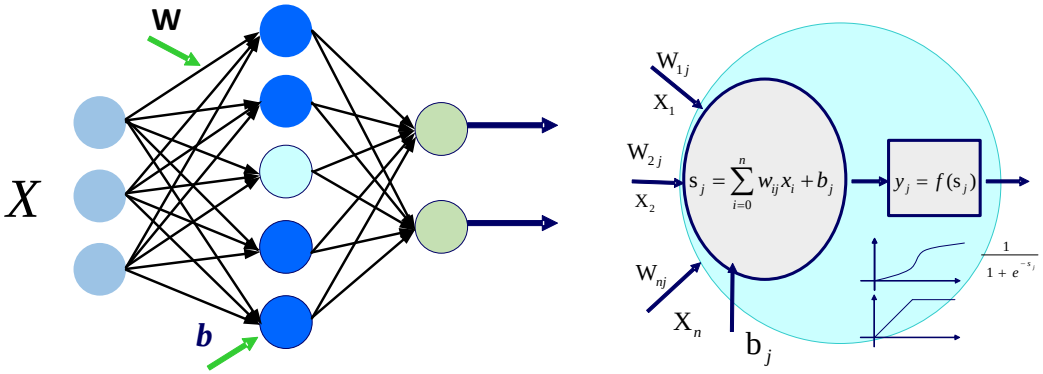


Figure 2.7: Feed forward neural network and activation neuron

the loss function. Fig. 2.7 shows a single hidden layer neural network and a activation neuron. Here, the input $\boldsymbol{X}$ is multiplied by the weight $\boldsymbol{W}$ and then performs the activation function in each neuron. We can further add hidden layers to form deep learning, which can learn hierarchal features from the training data. The training process can be performed by backward propagation to update the weight in each layer.

Recently, deep learning achieves the state-of-the-art results in many applications, such as speech recognition [46, 47] and image recognition [48]. It becomes the major data analytics technique. However, deep learning poses new grand challenges for data analytics on hardware, especially for IoT devices. Firstly, because of the adoption of deep learning, the number of neural network layers increases significantly leading to a very large model. Since most IoT devices are highly resource-constrained in terms of CPU computation, main memory and bandwidth, it becomes greatly needed to optimize the machine learning algorithm for a compact model. Secondly, most IoT based system requires a real-time and energy-efficient performance. To provide a real-time and high-quality performance on IoT devices, both neural network algorithms and hardware accelerator are required to re-examine to meet the emerging needs. Therefore, in the following three sections, we will discuss machine learning algorithms on IoT devices and machine learning hardware accelerators. More specifically, a distributed neural network is discussed followed by the machine learning accelerator. The neural network model optimization considering the IoT hardware requirements is also discussed in the end.

2.3.2 Distributed Machine Learning

Distributed machine learning refers to machine learning algorithms running on multi-nodes to improve performance and scale to larger input data size [1]. To perform

distributed machine learning, algorithms are required to convert from single-thread to multi-thread and a multi-node computation framework has to be developed. We will further discuss distributed machine learning algorithms from both algorithm parallelism and computation frameworks.

Distributed Algorithms As mentioned above, distributed machine learning algorithms are to develop highly parallel algorithms running on multi computational nodes. There are mainly two types of parallelism: data parallelism and model parallelism. Data parallelism is mainly used to partition the training data into several subsets and then perform the training on different machines in a parallel fashion as shown in Fig. 2.8(a). The training model in each computational node is the same. Then a center server recording all the parameters is required to synchronize all the local updates. Each local computational node will be refreshed by downloading new parameters. This method mainly has two drawbacks. Firstly, a centralized parameter server offsets the benefits of distributed computation due to the relative communication delay, especially for IoT system. Secondly, it requires a replica of machine learning model on each computational node, which may limit the machine learning model size due to the limited computational resource. On the other hand, a model parallelism based machine learning algorithms will partition the model into multiple sub-models. For each training sample, each sub-model will collaborate with each other to perform the optimization as shown in Fig. 2.8(b). However, this training method is very sensitive to communication delay and may result in training failure if one computational node is down. Furthermore, the intermediate data size is huge for training algorithms such as stochastic gradient descent (SGD) like algorithms. Therefore, a model parallelism based machine learning algorithm, which can minimize the communication without sacrificing the model size, is greatly needed.

To overcome the high communication cost, an ensemble learning based machine learning algorithm becomes a potential solution. Ensemble learning algorithm is to build a set of classifiers to improve the accuracy performance comparing to a single classifier. Each classier can be trained independently using the subset of the training data and then combined in a concrete way defined by ensemble learning. Thus, ensemble learning is widely applied to resource limited IoT devices, since each IoT device can train a relative weak classifier and then combine to have a better performance. Moreover, the data in the IoT network is naturally distributed in each sensor and such training method avoids the high communication overhead by building a local classifier. The advantages of ensemble learning on IoT network are summarized as follows.

- **Accuracy**: Using different training methods and then combining the prediction results can improve the overall accuracy. This is mainly because the merge of

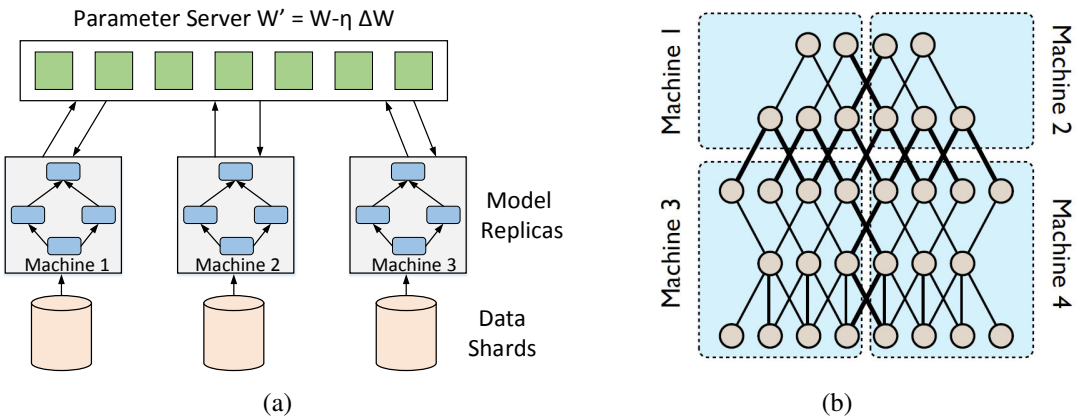


Figure 2.8: (a) Data parallelism by partitioning training data on different machines (b) Model parallelism by local machine collaboration for optimization[1]

different classifiers can compensate the inefficient characteristics for each other. Ideally, if we can have an independent classifier with accuracy more than 50%, the overall accuracy can approach 100% by increasing the number of such independent classifiers.

- **Communication efficiency**: The communication efficiency are twofold. Firstly, there is no need to upload the IoT collected data to the cloud. A local computational node can perform the training as a local classifier. Secondly, we can build multiple classifiers without communication with each other. This is especially important since stochastic gradient descent (SGD) based training method requires a huge communication bandwidth, which slows down the training process.

- **Scalability**: The ensemble learning makes use of the limited computational resource and can easily integrate more devices and models to improve its performance. It also overcomes the storage limitation by building small-sized classifiers and storing parameters locally in the distributed IoT devices.

Distributed Computing Framework The storage and computation of huge data size has always been a grand challenge for IoT system due to the limited hardware resource and real-time requirements. Many computing frameworks have been proposed for parallel and distributed computing. There are mainly three general frameworks, which range from the low-level framework that provides only the basic functionality to the high-level framework that provides automatic fault tolerance application programming interface (API). Therefore, we first introduce the three computing frameworks and then analyze the proper framework for the IoT system.

Message Passing Interface (MPI)[49] is a low-level computation framework designed for high performance with a standard message passing specification. MPI is small but able to support many applications with only 6 basic functions (Initialization, Exit, Send, Receive and etc.). Furthermore, it is scalable by providing point-to-pint communication and flexible without the need for rewriting parallel programs across platforms. However, due to its low-level nature, it requires careful programming and developers have to explicitly handle the data distributions, which increases the development time.

Hadoop [50] is an open source framework for processing large dataset proposed by Google. Hadoop is designed to connect many commodity computers to work in parallel to tackle a very large amount of data. A Hadoop job provides a *map* step and a *reduce* step. A *map* step will read a bunch of data and emit a series of key-value pairs whereas a *reduce* step takes the key value pairs and computes the reduced set of data. Hadoop provides an automatic fault-tolerance system, a distributed file system, and a simple programming abstraction that allows users to analyze petabyte-scale data across thousands of machines. However, Hadoop cannot natively or efficiently support iterative workflows. It requires the user to submit a single job for every iteration. Furthermore, intermediate results must be materialized to disk, causing the performance of iterative queries to suffer. It also requires heavy communication and synchronization between computational nodes causing the performance degradation.

Apache Spark [51] is a distributed framework that provides in-memory data processing engine with expressive development API. Spark is built using Hadoop MapReduce and extends the MapReduce paradigm by providing more types of computations, which includes iterative queries and stream processing. Additionally, by keeping the working dataset in memory, Spark provides 100 times faster in memory and 10 times faster when running on disk. However, similar to the limitation of Hadoop, the relatively large memory requirement for such distributed computational framework is not suitable for IoT devices. It recommends at least 8GB memory and 8 cores per machine [52], which is far beyond the resource available on IoT devices.

As aforementioned, considering the limited resources of various IoT devices, we choose MPI as the distributed computing framework due to its light-weight, scalable and flexible characteristics.

2.3.3 Machine Learning Accelerator

Due to the low-power and real-time data analytics requirement on IoT systems, there is an emerging need to develop machine learning accelerators for both training and inference process. The current state of machine learning is mainly dominated by general

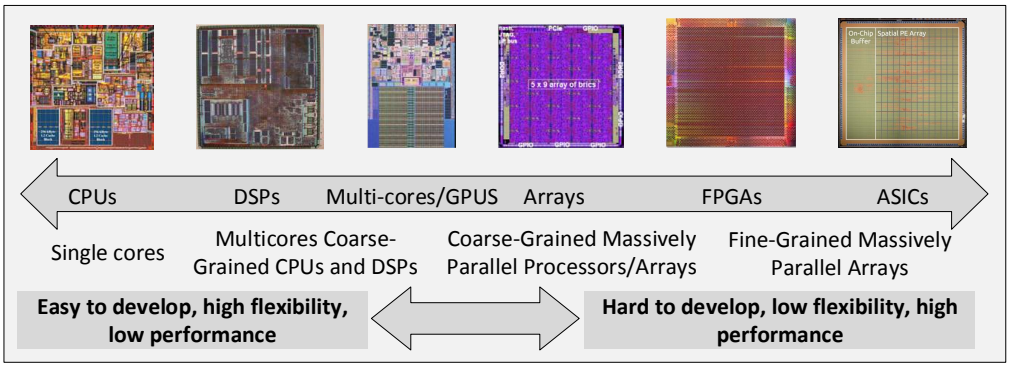


Figure 2.9: Trade-off between performance, flexibility and ease to develop on various hardware platform

purpose graphic processor unit (GPGPU). GPU has orders of magnitude of more computing cores than central general processor unit (CPU), which make it faster to perform parallel computation of machine learning algorithms [53]. However, GPU based implementation consumes significant power (performance per watt), which is not applicable to IoT systems such as smart homes. Therefore, there is an increasing need to develop an energy-efficient accelerator for fast inference and training process.

Considering a hardware accelerator, there is a wide range of hardwares featuring different trade-offs between performance, ease to develop and flexibility. As shown in Fig. 2.9, we can show two extreme points for hardware computation platform. At one end, a single core CPU is easy to use and develop but suffers from low performance. At the other end, application specific integrated circuit (ASIC) provides high performance at the cost of high development difficulty and inflexibility. Field-programmable gate array (FGPA) is a compromise of these two methods, which provides a fine-tuned reconfigurable architecture and energy-efficient performance. From the circuit level design, FPGA can implement sequential logics through the use of flip-flops (FF) and combinational logic through the use of look-up tables [53]. By performing timing analysis, pipeline stage can be inserted to improve the clock speed. From the system level design, FPGA features high level synthesis and can convert C-Program language to synthesizable hardware description language (HDL) to accelerate the development process. Moreover, the new released FPGA shows a system-on-chip (SoC) design approach with an embedded ARM core. Such development trend of FPGA provides a high flexibility of neural network architecture on FPGA as well as low-power performance for targeted applications.

The benefits of adopting hardware accelerator in IoT systems are twofold: lower latency and higher energy efficiency. Firstly, hardware accelerator performs faster than

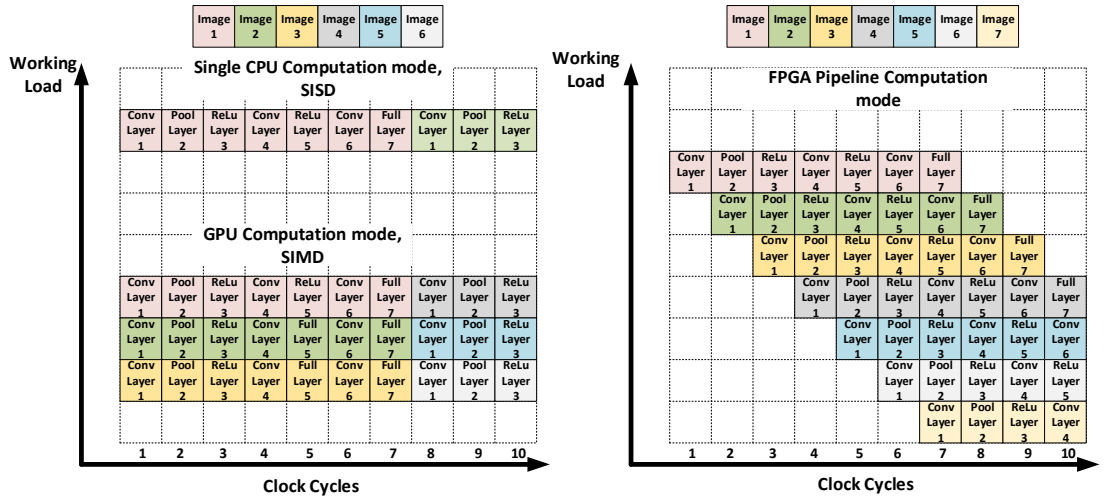


Figure 2.10: CPU, GPU and FPGA computational mode and parallelism for a 7 layer neural network (Conv layer - Pooling layer - ReLu layer - Conv layer - ReLu layer - Conv layer - Full layer)[4]

general-purpose system and the silicon size is also much smaller. For real-time applications such as video surveillance and face recognition, a hardware accelerator can detect the face in the videos leading to high-level security. Another example is the human action recognition. A real-time human action recognition system is required to understand human actions and then perform the desired operations. Secondly, hardware accelerator is much more energy efficient than general processors (CPU, GPU). For IoT network intrusion detection, the network load is very heavy and communication speed is high. Using general-purpose system to monitor network traffic is slow and energy consuming. Instead, a FPGA based hardware accelerator is reconfigurable and specially designed to quickly detect abnormal network traffic with low power consumption.

The performance of machine learning accelerators depends on the joint algorithm and architecture optimization. The algorithm optimization is designed to reduce the computation cost by adopting bit-width reduction, model compression and sparsity. The model optimization will be discussed in details in Chapter 2.3.4. In this chapter, we will focus on the architecture level design, which consists of computing engine design and memory system design.

For the computing engine optimization, many works explore the parallelism of neural network from both data parallelism and pipeline parallelism to improve the overall throughput. [55] proposed a relative complex neuron processing engine (NPE), which consists of multiplication and accumulation (MAC) blocks, activation function blocks and max pooling blocks. This is a mix of data parallelism and pipeline parallelism. In

[4]Details on the definition of each layer(Conv layer, ReLu Layer and Full layer) can be found at [54]

this design, the same kernel filter data is shared for different MAC computations in one NPE and different kernels are applied to different NPEs to utilize the input image data. [56] adopted a highly-parallel spatial architecture of simple processing elements based on data flow processing. Each processing element is featured with flexible autonomous local control and local buffer. The partial sum is computed by processing elements (PEs) in a spatial array and stores in the local memory for final summation. The summation result is used for the next layer computation. In addition, the optimization of computation from the mathematics perspective can also be adopted. [57] proposed to apply Strassen algorithm to reduce the computational workload of matrix multiplication. This algorithm reduces the number of multiplication at the cost of increasing matrix additions. It is reported that an up to 47 % computation load reduction can be achieved for certain layer. However, this requires more logic control and significantly more memory compared to the naive matrix-vector multiplication.

For the memory system optimization, the major technique is to minimize accesses from the expensive levels of the memory hierarchy. Techniques such as local buffer, tiling and data reuse are developed to minimize the data movement. The tiling and data reuse technique can effectively cut down the memory traffic but a very careful design of data movement is required. A local storage buffer on PE is a dedicated buffer for PE, which is designed to maximize the data reuse, but becomes infeasible when the memory size of intermediate results is huge. The same concerns happen to the on-chip memory, where model parameters are huge to store on chip. Many works have been studied to adopt the on-chip memory but most works design the on-chip memory by careful data movement and data compression techniques. [58] proposed a tiled constitutional layer to adopt on-chip memory and reduce the external memory access. A roofline model is also developed to optimize the tile size. [56] adopted the run-length compression technique to reduce the image bandwidth requirement.

In addition to CMOS technology, memristive devices are also a potential hardware solution to machine learning accelerator. The term “Memristive” comes form Leon Chua in the 70’s [59]. Here, we mainly focus on Resistive RAMs (RRAMS). Such device is a two-terminal device with 2 non-volatile states: high resistance state (HRS) and low resistance state (LRS). The state of RRAM is determined by the write voltage on its two terminals. It is most stable in bistate, where high resistance state (HRS) and low resistance state (LRS) are determined by the polarity of write voltage. Two computation schemes on RRAM have been proposed in literature, which are analog RRAM computation [60] and digital RRAM computation [61]. [60] proposed the analog matrix-vector multiplication with consideration of non-ideal factors of device and circuit. Furthermore, [62] applied RRAM technology to spike neural network to perform real-time classification.

However, analog matrix multiplication suffers from the lack of accuracy, device variations and large power consumption of the analog-to-digital conversion. On the other hand, [61] proposed a digital RRAM in-memory computation. Later, [63] applied this matrix multiplication for on-line machine learning. Therefore, in this thesis, we mainly apply digital RRAM based computation to machine learning accelerator for low-power IoT applications.

2.3.4 Machine Learning Model Optimization

Recently, neural network compression has gained much attention, motivated by mapping deep neural networks to resource limited hardwares such as IoT devices. Many researches have been conducted on neural network compression, which can be summarized as hashing technique [64, 65], weights quantization [12, 66, 67], connection pruning [8, 68], distilled knowledge [69, 70] and matrix decomposition [71, 72]. We will elaborate each technique in more details.

Hashing technique [64, 65] is to apply hash mapping function to compress parameters. [64] applied a hash function to group network parameters into hash buckets randomly as shown in Fig. 2.11(a). Fig. 2.11(a) shows how the model is compressed from 9 parameters to 3 parameters ($3\times$ compression rate). The training process follows the standard backward propagation but with hashing function.

Weight quantization is one of the most common approaches to reduce the machine learning model size. [67] proposed to convert the floating point weight to a fixed point representation. By analyzing the neural network weight distribution and adopting dynamic fixed point representation, it achieves more than 20% reduction of the model size. Furthermore, [66] proposed a quantized neural network obtained during training with constraints of binarized weight. It performs very well for small dataset such as MNIST or CIFAR but suffers significant accuracy loss. [12] extended this to a binarized neural network, where all the weights are binarized during the training process. Fig. 2.11(b) shows the compression of the neural network weight with $32\times$ compression from 32 bit floating point to 1 bit binary weight.

The connection pruning method is to delete unimportant connections in the neural networks. [68] proposed a node pruning method to compress the neural network by singular value decomposition (SVD) method. It suggests that a very small singular value indicates redundancy or unimportant connection in the neural networks and hence can be pruned. [8] proposed a deep compression method by combining node pruning, weight quantizations and Huffman coding method. This method achieves around $100\times$ compression rate but requires recursively training for each procedure. Since training a

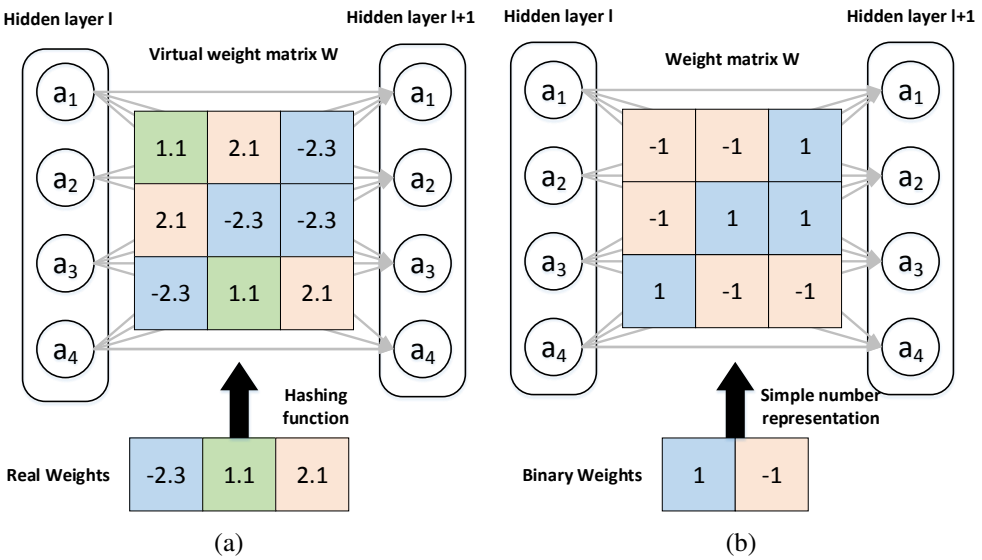


Figure 2.11: Neural network compression (a) Hashing trick; (b) binarized neural network

deep neural network takes days or weeks, such method may not be applicable for large neural networks.

Distilling knowledge is to make use of the knowledge from a big well trained neural network to train an ensemble of smaller neural networks [70]. [69] showed that we can compress the knowledge from a big neural network into a smaller neural network by generating pseudo training data. Furthermore, [70] showed that by using the softmax output from the big cumbersome model as the soft target distribution, we can train a much smaller neural network with significant compression rate. However, this method still suffers a relatively large loss for model compression.

The most relevant works to our thesis are based on low-rank approximation of weight matrices [72, 73]. For example, a low-rank decomposition of the neural network weight matrix:

$$\boldsymbol{W} = \boldsymbol{A} \times \boldsymbol{B} \tag{2.1}$$

where $\boldsymbol{W} \in \mathbb{R}^{m \times n}$, $\boldsymbol{A} \in \mathbb{R}^{m \times r}$ and $\boldsymbol{B} \in \mathbb{R}^{r \times n}$. As such, one can effectively compress the weight matrix from mn to $mr + nr$ given a small rank r.

A tensor decomposition is a generalized low-rank matrix decomposition. By representing dense data (matrix) in high dimensional space (tensor) with a low rank, even higher network compression rate can be achieved [72]. However, the work in [72] requires complicated training for multi-layer network using tensorized backward propagation (BP), which is hard to implement and slow to train. In this work, we propose a layer-wise training method based on an efficient modified alternating least-squares

method (MALS [74]). This is fundamentally different from the training method in [72]. Moreover, a left-to-right sweeping on tensor cores is further applied to efficiently reduce the rank, leading to a higher network compression. Nevertheless, a non-uniform quantization optimization is also applied on tensor cores for the simplified numeric representation of weights under controlled accuracy. We will illustrate this in Chapter 4 in more details.

2.4 Summary

In this chapter, we discuss IoT based smart building with machine learning algorithms. By adopting machine learning algorithms, the building management can sense the environmental data, analyze it and then perform optimal decisions. More specifically, this chapter discusses the literature review of indoor positioning system, energy management system and network security system in IoT systems. The machine learning algorithm can effectively understand the data and thereby assist occupants living as well as save energy. Moreover, we argue that by performing computation locally, we can perform real-time data analytics and protect the data privacy.

Furthermore, we discuss the basics of machine learning techniques to accelerate machine learning process. More specifically, distributed machine learning, machine learning accelerators and machine learning model compression are discussed in details. Distributed machine learning converts a single-thread algorithm into a multi-thread algorithm running in distributed computational nodes. We propose an ensemble learning algorithm to build a set of classifiers running on IoT devices for high quality performance. Moreover, the total design flow for high throughput machine learning accelerator is summarized in Fig. 2.12. Model compression such as hashing net, node pruning and neural network weight decomposition is the major technique to reduce the model size. Our contribution on model compression is to propose a tensor-train based neural network weight decomposition method to reduce the model size. A dynamic quantization method is also proposed for weight quantization. For the machine learning accelerator design, data reuse, tiling and on-chip memory are the major techniques to improve the processing speed for higher throughput. Our designed accelerator also utilizes these design techniques to improve the throughput. In addition, we discuss the new emerging device, which is the resistive random-access memory (RRAM) as the potential computational element for machine learning accelerators.

In the following three chapters, we will discuss three machine-learning solvers on

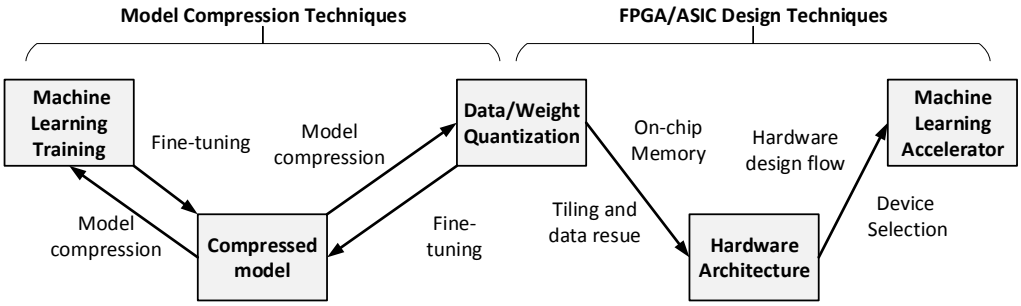


Figure 2.12: Machine learning accelerator design flow with model compression and data quantization

IoT devices, which are the least-squares-solver, the tensor-solver and the distributed-solver. The least-squares-solver is a direct solver, which is designed to tackle small-sized classification and regression problems. The tensor-solver is designed towards deep neural networks using tensor compression. The distributed-solver is further proposed to improve performance for large IoT networks. These proposed solvers are designed to perform machine learning locally on IoT devices with lower latency, higher energy-efficiency, better privacy and larger scalability. The main applications can be real-time data analytics on IoT devices such as energy management system and network intrusion detection system.

Chapter 3

Least-Squares-Solver for Shallow Neural Network

3.1 Introduction

The machine-learning based data analytics has been widely adopted for artificial intelligence based cloud applications [75, 76]. Given a large amount of cloud data, deep neural network (DNN) based machine learning algorithms [75] are first practiced off-line for training to determine the DNN network weights; and then the trained DNN network is further deployed online to perform inference using the new input of data. The current training method is mainly based on an iterative backward propagation method [77], which has long latency (usually days and weeks) running on data servers. However, with the emergence of autonomous vehicles, unmanned aerial vehicle, robotics [78] and IoT systems, there is a huge demand to analyze the real-time sensed data with small latency in data analytics. Usually, the problem size of the real-time sensed data is not like the size of cloud data. As such, an online machine learning algorithm based real-time data analytics becomes an emerging need. One needs to either develop a fast training algorithm or utilize a hardware-based accelerator [58, 79, 80, 81].

The training process of a neural network is to approximate nonlinear mapping from the input samples to the desired output with sufficient generalities. Iterative solvers such as backwards propagation are widely used for training large-sized data at cloud but may not be necessary for a limited problem size of data during the real-time data analytics. Moreover, since iterative solvers improve the solution in each iteration, the number of iterations and the computation load are usually non-predictable, and thereby requires human interference to determine the optimal solution. On the other hand, direct solvers can automatically compute the result to equip IoT devices with intelligence. The recent

extreme learning machine (ELM) based approach [82, 83] has shown that a shallow neural network (single hidden layer neural network) can provide acceptable accuracy with significant training time reduction. However, this method relies on a pseudo-inversion of matrix to solve the least-squares (LS) problem, which requires relatively high computational cost.

Currently, hardware-based accelerator is designed and practiced to handle the high complexity of machine learning. One needs to develop a dedicated hardware architecture with built-in pipeline and parallelism of data processing. Moreover, since many applications can share one similar neural network architecture, a parameterized and reconfigurable accelerator targeting a number of applications becomes appealing. Many recent hardware-based accelerator works are however mainly designed for the inference (or classification) stage [58, 84] with little exploration in training, which is the bottleneck in real-time data analytics. Therefore, it is an unique opportunity to develop an energy-efficient hardware accelerator considering both training and inference as the common learning engine for IoT systems. In this chapter, we develop a least-squares (LS) based machine learning algorithm with two hardware implementations. More specifically, this chapter investigates these two implementations.

- A single-precision floating-point hardware-based accelerator for both training and inference phases on FPGA is proposed. An online machine learning is developed using a regularized least-squares-solver with incremental square-root-free Cholesky factorization. A corresponding scalable and parameterized hardware realization is developed with a pipeline and parallel implementation for both regularized least-squares-solvers and also matrix-vector multiplication. With the high utilization of the FPGA hardware resource, our implementation has 32 processing elements (PEs) running in parallel at 40-MHz. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable energy load forecasting accuracy with an average speed-up of 4.56× and 89.05×, when compared to x86 CPU and ARM CPU for inference respectively. Moreover , 450.2×, 261.9× and 98.92× energy saving can be achieved comparing to x86 CPU, ARM CPU and GPU respectively.

- A 3D multi-layer CMOS-RRAM accelerator architecture for incremental machine learning is presented. By utilizing an incremental least-squares-solver, the whole training process can be mapped to the 3D multi-layer CMOS-RRAM accelerator with significant speed-up and energy-efficiency improvement. Experiment results using the benchmark CIFAR-10 show that the proposed accelerator has

2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC hardware implementation; and 14.94× speed-up, 447.17× energy-saving and around 164.38× area-saving compared to CPU software implementation. Compared to GPU implementation, our work shows 3.07× speed-up and 162.86× energy-saving.

The rest of this chapter is organized as follows. Chapter 3.2 discusses an optimized Cholesky decomposition based least-squares algorithm for single hidden layer neural network. The hardware implementation of the optimized learning algorithm is discussed in Chapter 3.3. Experimental results for the CMOS and RRAM based implementation are shown in Chapter 3.4 with conclusion drawn in Chapter 3.5.

3.2 Algorithm Optimization

3.2.1 Preliminary

Neural network (NN) is a family of network models inspired by biological neural network to build the link for a large number of input-output data pairs. It typically has two computational phases: training phase and inference phase.

- In the training phase, the weight coefficients of the neural network model are first determined using training data by minimizing the squares of error difference between trial solution and targeted data in a so-called ℓ_2-norm method.

- In the inference phase, the neural network model with determined weight coefficients is utilized for classification or regression given the new input of data.

Table 3.1: A list of parameters definitions in machine learning

Parameter	Elements	Definitions
$\boldsymbol{X}$	$[x_{11}, x_{12}, x_{13}, ..., x_{Nn}]$	A set of n dimension data in N training samles
$\boldsymbol{T}$	$[t_{11}, t_{12}, t_{13}, ..., t_{Nm}]$	A set of m target classes in N training samples
$\boldsymbol{H}$	$[h_{11}, h_{12}, h_{13}, ..., h_{NL}]$	A set of L hidden nodes in N training samples
$\boldsymbol{A}$	$[a_{11}, a_{12}, a_{13}, ..., a_{nL}]$	Input weight matrix between $\boldsymbol{X}$ and $\boldsymbol{H}$
$\boldsymbol{\Gamma}$	$[\gamma_{11}, \gamma_{12}, \gamma_{13}, ..., \gamma_{Lm}]$	Output weight matrix between $\boldsymbol{H}$ and $\boldsymbol{T}$
$\boldsymbol{Y}$	$[y_1, y_2, y_3, ..., y_m]$	A set of m model outputs
$\boldsymbol{\beta}$	N.A.	Learning rate set by designers

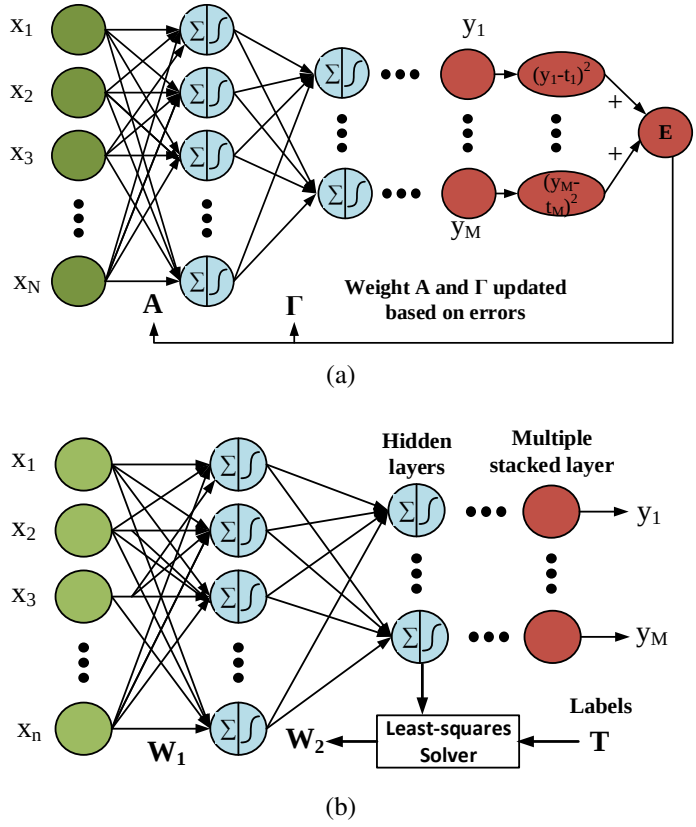


Figure 3.1: Trainings of neural network: (a) backward propagation; and (b) direct solver

Formally, the detailed description of each parameter is summarized in Table 3.1. Given a neural network with n inputs and m outputs as shown in Fig. 3.1, a dataset $(x_1,t_1),(x_2,t_2),...,(x_N,t_N)$ is composed of paired input data $\boldsymbol{X}$ and training data $\boldsymbol{T}$ with N number of training samples, n dimensional input features and m classes. During the training, one needs to minimize the ℓ_2-norm error function with determined weights: $\boldsymbol{A}$ (at input layer) and $\boldsymbol{\Gamma}$ (at output layer):

$$E = ||\boldsymbol{T} - F(\boldsymbol{A},\boldsymbol{\Gamma},\boldsymbol{X})||_2 \tag{3.1}$$

where $F(\cdot)$ is the mapping function from the input to the output of the neural network.

The output function of this neural network classifier is

$$\boldsymbol{Y} = F(\boldsymbol{A},\boldsymbol{\Gamma},\boldsymbol{X}), \;\; \boldsymbol{Y}^T = [y_1 \; y_2 \; ... \; y_N] \tag{3.2}$$

where $\boldsymbol{Y} \in \mathbb{R}^{N\times m}$. $\boldsymbol{Y}^T$ refers to the transpose of $\boldsymbol{Y}$ and y_i is a column vector. N represents the number of inference samples. For the i-th inference sample, the index of maximum value y_i is found and identified as the predicted class.

Backward Propagation for Training

The first method to minimize the error function E is the backward propagation (BP) method. As shown in Fig. 3.1(a), the weights are initially guessed for forward propagation. Based on the trial error, the weights are adjusted by the derivatives of weights as follows

$$\nabla E = (\frac{\partial E}{\partial a_{11}}, \frac{\partial E}{\partial a_{12}}, \frac{\partial E}{\partial a_{13}} ... \frac{\partial E}{\partial a_{nL}}) \tag{3.3}$$

where n is the dimension of input data and L is the number of hidden nodes. For the input layer, each weight can be updated as

$$a_{dl} = a_{dl} - \beta * \frac{\partial E}{\partial a_{dl}},\ d = 1,2,...,n,\ l = 1,2,...,L \tag{3.4}$$

where β is the learning constant that defines the step length of each iteration in the negative gradient direction. Note that the BP method requires to store the derivatives of each weight. It is expensive for hardware realization. More importantly, it may be trapped on local minimal with long convergence time. Hence, the BP based training is usually performed off-line and has large latency when analyzing the real-time sensed data.

Direct Solver for Training

One can directly solve the least-squares problem using the direct-solver of the ℓ_2-norm error function E [82, 85, 86]. As shown in Fig. 3.1(b), the input weight $\boldsymbol{A}$ can be first randomly assigned and one can directly solve output weight $\boldsymbol{\Gamma}$.

We first find the relationship between the hidden neural node and input training data as

$$\boldsymbol{preH} = \boldsymbol{XA} + \boldsymbol{B},\ \boldsymbol{H} = Sig(\boldsymbol{preH}) = \frac{1}{1 + e^{-\boldsymbol{preH}}} \tag{3.5}$$

where $\boldsymbol{X} \in \mathbb{R}^{N \times n}$. $Sig(\boldsymbol{preH})$ refers to the element-wise sigmoid operation of matrix $\boldsymbol{preH}$. $\boldsymbol{A} \in \mathbb{R}^{n \times L}$ and $\boldsymbol{B} \in \mathbb{R}^{N \times L}$ are randomly generated input weight and bias formed by a_{ij} and b_{ij} between $[-1,1]$ respectively. N and n are the training size and the dimension of training data respectively. The output weight $\boldsymbol{\Gamma}$ is computed based on pseudo-inverse $(L < N)$:

$$\boldsymbol{\Gamma} = (\boldsymbol{H}^T \boldsymbol{H})^{-1} \boldsymbol{H}^T \boldsymbol{T} \tag{3.6}$$

However, performing pseudo-inverse is expensive for hardware realization.

Algorithm 1 Learning Algorithm for Single Layer Network

1: Randomly assign hidden-node parameters
2: (a_{ij}, b_{ij}), $a_{ij} \in \boldsymbol{A}$, $b_{ij} \in \boldsymbol{B}$
3: Calculate the hidden-layer pre-output matrix $\boldsymbol{H}$
4: $\boldsymbol{preH} = \boldsymbol{XA} + \boldsymbol{B}$, $\boldsymbol{H} = 1/(1+e^{-\boldsymbol{preH}})$
5: Calculate the output weight
6: $\boldsymbol{\Gamma} = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{T}$
7: Calculate the training error *error*
8: $error = ||\boldsymbol{T} - \boldsymbol{H\Gamma}||$
9: IF ($L \leq L_{max}$ and $e > \varepsilon$)
10: Increase number of hidden node
11: $L = L + 1$, repeat from Step 1
12:
13: ENDIF

The comparison between backwards propagation (BP) and direct solvers can be summarized as follows. BP is an iterative solver, which is relatively simple implementation by gradient descent objective function with good performance. However, it suffers from the long training time and may get stuck in the local optimal point. As mentioned in Chapter 3.1, iterative solvers are more applicable to train large-sized data at cloud scale but may not be necessary for a limited problem size of data during the real-time data analytics. Moreover, since iterative solvers improve the solution in each iteration, the number of iterations is usually non-predictable, and thereby requires human interference to determine the optimal solution. On the other hand, direct solver can learn very fast without human interference, but pseudo-inverse is too expensive for calculation. Therefore, solving ℓ_2-norm minimization efficiently becomes the bottleneck of the training process. Algorithm 1 shows the training process of single hidden layer neural network. Note that the number of hidden nodes L is also auto-configured in Algorithm 1.

3.2.2 Incremental Least-Squares Solver

From Algorithm 1, we can have two major observations. Firstly, the number of hidden nodes is increasing sequentially to meet accuracy requirement. The previous results should be utilized when the number of hidden nodes is increased. Secondly, the key difficulty of solving training problem is the least-squares problem. This could be solved by using singular vector decomposition (SVD), Gram-Schmidt Decomposition (QR) and Cholesky decomposition. The computational cost of SVD, QR and Cholesky decomposition are $O(4NL^2 - \frac{4}{3}L^3)$, $O(2NL^2 - \frac{2}{3}L^3)$ and $O(\frac{1}{3}L^3)$ respectively [87]. Therefore, we use Cholesky decomposition to solve the least-squares problem. Here, we show how to

incrementally solve the least-squares problem by re-using previous Cholesky decomposition in the neural network training process. Below is the incremental equation when we increase the number of hidden nodes:

$$\boldsymbol{preH} = \boldsymbol{XA'} + \boldsymbol{B'} = (\boldsymbol{X} * [\boldsymbol{A}, \boldsymbol{a}] + [\boldsymbol{B}, \boldsymbol{b}]) = [(\boldsymbol{XA} + \boldsymbol{B}),\ (\boldsymbol{Xa} + \boldsymbol{b})] \tag{3.7}$$

where $\boldsymbol{A'}$ and $\boldsymbol{B'}$ are the new input weights with increased size from $\boldsymbol{A}$ and $\boldsymbol{B}$. For the simplicity, we use L to represent the number of hidden nodes and denote h_L as the new added column generated from activation of $(\boldsymbol{Xa} + \boldsymbol{b})$. The multiplication of activation matrix for the neural network is

$$\boldsymbol{H}_L^T \boldsymbol{H}_L = \begin{bmatrix} \boldsymbol{H}_{L-1}\ h_L \end{bmatrix}^T \begin{bmatrix} \boldsymbol{H}_{L-1}\ h_L \end{bmatrix} = \begin{pmatrix} \boldsymbol{H}_{L-1}^T \boldsymbol{H}_{L-1} & \boldsymbol{v}_L \\ \boldsymbol{v}_L^T & g \end{pmatrix} \tag{3.8}$$

The Cholesky decomposition can be expressed as

$$\boldsymbol{H}_L^T \boldsymbol{H}_L = \boldsymbol{Q}_L \boldsymbol{D}_L \boldsymbol{Q}_L^T \tag{3.9}$$

where $\boldsymbol{Q}_L$ is a lower trangular matrix with diagonal elements $q_{ii} = 1$ and $\boldsymbol{D}_L$ is a positive diagonal matrix. Such method can maintain the same space as Cholesky factorization but avoid extracting the square root as the square root of $\boldsymbol{Q}_L$ is resolved by diagonal matrix $\boldsymbol{D}_L$ [88]. Here, we use $\boldsymbol{H}_L$ to represent the matrix with L number of hidden neural nodes. Similarly, $\boldsymbol{H}_L^T \boldsymbol{H}_L$ can be decomposed as follows

$$\begin{aligned} \boldsymbol{H}_L^T \boldsymbol{H}_L &= \begin{bmatrix} \boldsymbol{H}_{L-1}\ h_L \end{bmatrix}^T \begin{bmatrix} \boldsymbol{H}_{L-1}\ h_L \end{bmatrix} \\ &= \begin{pmatrix} \boldsymbol{H}_{L-1}^T \boldsymbol{H}_{L-1} & \boldsymbol{v}_L \\ \boldsymbol{v}_L^T & g \end{pmatrix} \end{aligned} \tag{3.10}$$

where $(\boldsymbol{v}_L, g)$ is a new column generated from new data compared to $\boldsymbol{H}_{L-1}^T \boldsymbol{H}_{L-1}$. Furthermore, we can find

$$\begin{aligned} &\boldsymbol{Q}_L \boldsymbol{D}_L \boldsymbol{Q}_L^T \\ &= \begin{pmatrix} \boldsymbol{Q}_{L-1} & 0 \\ \boldsymbol{z}_L^T & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{D}_{L-1} & 0 \\ 0 & d \end{pmatrix} \begin{pmatrix} \boldsymbol{Q}_{L-1}^T & \boldsymbol{z}_L \\ 0 & 1 \end{pmatrix} \end{aligned} \tag{3.11}$$

As a result, we can easily calculate the vector $\boldsymbol{z}_L$ and scalar d for Choskey factorization as

$$\boldsymbol{Q}_{L-1} \boldsymbol{D}_{L-1} \boldsymbol{z}_L = \boldsymbol{v}_L,\ d = g - \boldsymbol{z}_L^T \boldsymbol{D}_{L-1} \boldsymbol{z}_L \tag{3.12}$$

Algorithm 2 Square-root-free ℓ_2-norm solution

Input: Activation matrix $\tilde{\boldsymbol{H}}_L$, target matrix $\tilde{\boldsymbol{T}}$ and number of hidden nodes L
Output: Neural Network output weight x

1: Initialize $r_0 = \tilde{\boldsymbol{T}}$, $\Lambda_0 = \emptyset$, $d = \boldsymbol{0}$, $x_0 = \boldsymbol{0}$, $l = 1$,
2: While $||r_{l-1}||_2^2 \leq \varepsilon^2$ or $l \leq L$
3: $c(l) = h_l^T r_{l-1}$, $\Lambda_l = \Lambda_{l-1} \cup l$
4: $\boldsymbol{v}_l = \tilde{\boldsymbol{H}}_{\Lambda_l}^T h_l$
5: $\boldsymbol{Q}_{l-1} w = \boldsymbol{v}_l(1 : l-1)$. $\boldsymbol{z}_l = w./diag(\boldsymbol{D}_{l-1})$
6: $d = g - \boldsymbol{z}_l^T w$
7: $\boldsymbol{Q}_l = \begin{bmatrix} \boldsymbol{Q}_{l-1} & \boldsymbol{0} \\ \boldsymbol{z}_l^T & 1 \end{bmatrix}$, $\boldsymbol{D}_l = \begin{bmatrix} \boldsymbol{D}_{l-1} & \boldsymbol{0} \\ \boldsymbol{0} & d \end{bmatrix}$, where $\boldsymbol{Q}_1 = 1, \boldsymbol{D}_1 = h_1 * h_1^T$
8: $\boldsymbol{Q}_l^T x_{tp} = \begin{bmatrix} \boldsymbol{0} \\ c(l)/d \end{bmatrix}$
9: $x_l = x_{l-1} + x_{tp}$, $r_l = r_{l-1} - \tilde{\boldsymbol{H}}_{\Lambda_l} x_{tp}$, $l = l + 1$
10: END While

where $\boldsymbol{Q}_L$ and $\boldsymbol{v}_L$ is known from (3.10), which means that we can continue to use previous factorization result and only update the corresponding part. Algorithm 2 shows more details on each step. Note that Q_1 is 1 and D_1 is $\boldsymbol{H}_1^T \boldsymbol{H}_1$.

The optimal residue for the least-squares problem $||\boldsymbol{H}\boldsymbol{\Gamma} - \boldsymbol{T}||_2$ is defined as r:

$$r = \boldsymbol{T} - \boldsymbol{H}\boldsymbol{\Gamma}_{ls} = \boldsymbol{T} - \boldsymbol{H}((\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{T}) \tag{3.13}$$

As such, r is orthogonal to $\boldsymbol{H}$ since the projection of r to $\boldsymbol{H}$ is

$$< r, \boldsymbol{H} >= \boldsymbol{H}^T(\boldsymbol{T} - \boldsymbol{H}\boldsymbol{\Gamma}_{ls}) = 0 \tag{3.14}$$

Similarly, for every iteration of Cholesky decomposition, x_{L-1} is the least-squares solution of $\boldsymbol{T} = \boldsymbol{H}_{\Lambda_{L-1}} * \boldsymbol{\Gamma}$ with the same orthogonality principle, where Λ_l is the selected column sets for matrix $\boldsymbol{H}$. Therefore, we have

$$\begin{aligned} \boldsymbol{T} &= r_{L-1} + \boldsymbol{H}_{\Lambda_{L-1}} * x_{L-1} \\ \boldsymbol{H}_{\Lambda_L}^T \boldsymbol{H}_{\Lambda_L} x_L &= \boldsymbol{H}_{\Lambda_L}^T (r_{L-1} + \boldsymbol{H}_{\Lambda_{L-1}} * x_{L-1}) \end{aligned} \tag{3.15}$$

where x_{L-1} is the least-squares solution in the previous iteration. By utilizing superposition property of linear systems, we can have

$$\begin{aligned} \begin{bmatrix} \boldsymbol{H}_{\Lambda_l}^T \boldsymbol{H}_{\Lambda_l} x_{tp1} \\ \boldsymbol{H}_{\Lambda_l}^T \boldsymbol{H}_{\Lambda_l} x_{tp2} \end{bmatrix} &= \begin{bmatrix} \boldsymbol{H}_{\Lambda_l}^T r_{L-1} \\ \boldsymbol{H}_{\Lambda_l}^T \boldsymbol{H}_{\Lambda_{L-1}} * x_{L-1} \end{bmatrix} \\ x_L = x_{tp1} + x_{tp2} &= x_{tp1} + x_{L-1} \end{aligned} \tag{3.16}$$

where the second row of equation has a trivial solution of $[x_{L-1}\ 0]^T$. Furthermore, this indicates that the solution of x_L is based on x_{L-1} and only x_{tp} is required to be computed out from the first row of (3.16), which can be expanded as

$$\boldsymbol{H}_{\Lambda_l}^T \boldsymbol{H}_{\Lambda_l} x_{tp1} = \begin{bmatrix} \boldsymbol{H}_{\Lambda_{L-1}}^T r_{L-1} \\ h_L^T r_{L-1} \end{bmatrix} = \begin{bmatrix} 0 \\ h_L^T r_{L-1} \end{bmatrix} \tag{3.17}$$

Due to the orthogonality between the optimal residual $\boldsymbol{H}_{\Lambda_{L-1}}$ and r_{L-1}, the dot product becomes 0. This clearly indicates that the solution x_{tp1} is a sparse vector with only one element. By substituting square-root-free Cholesky decomposition, we can find

$$\boldsymbol{Q}^T dx_{tp} = h_L^T r_{L-1} \tag{3.18}$$

where x_{tp} is the same as x_{tp1}. The other part of Cholesky factorization $\boldsymbol{Q}$ for multiplication of x_{tp1} is always 1 and hence is eliminated. The detailed algorithm including Cholesky decomposition and incremental least-squares is shown in Algorithm 2.

For the training problem of N number of training samples, with feature size n and L number of hidden nodes, the least-squares problem becomes to minimize $\|\boldsymbol{HW} - \boldsymbol{T}\|_2$, where $\boldsymbol{H}$ is a $N \times L$ matrix and $\boldsymbol{W}$ is a $L \times m$ matrix and $\boldsymbol{T}$ is $N \times m$. Here, the number of training samples N is much larger than L, which makes the problem become an over determined equation. The solution is $(\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{T}$, provided $\boldsymbol{H}$ has full column rank. For the multiplication $\boldsymbol{H}^T\boldsymbol{H}$, the computation complexity is $O(NL^2)$. The matrix inversion is $O(L^3)$.

Although from the complexity analysis the matrix multiplication dominates, the hardware operation complexity of matrix inversion is more complicated to accelerate. The multiplication computation can be easily accelerated by designing more parallel processing elements, which can linearly reduce the computation time. Given P parallel processing elements, the computational complexity can be reduced to $O(NL^2/P)$. Instead, the hardware operation complexity of matrix inversion is very high, which is very difficult to perform parallel computation on hardware accelerator. In the proposed algorithm 2, Step 4 and 6 are the vector-vector multiplication, which can be accelerated by the parallel computation. As such, the parallel processing elements for vector multiplication is more utilized. Step 5 and 8 are for the forward and backward substitution, which can be computed by reconfiguring PEs. The proposed algorithm 2 is using incremental Cholesky decomposition, which means only one loop in Algorithm 2 is performed. As such, the computation complexity is reduced to $O(L^2)$. Moreover, by utilizing Cholesky decomposition and incremental least-squares techniques, the hardware operation is reduced to only 4 basic linear algebra operations to improve PEs utilization

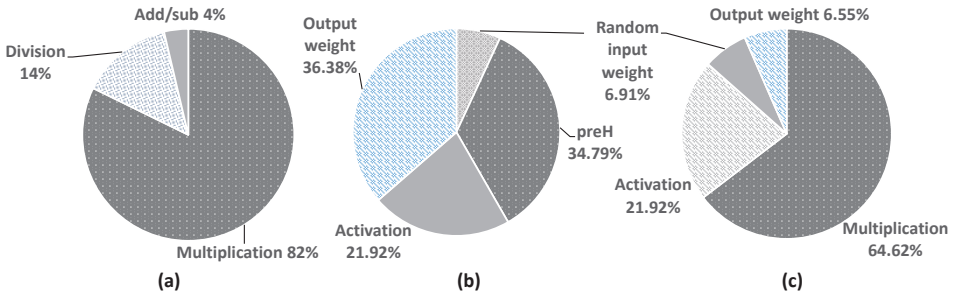


Figure 3.2: (a) Time consumption breakdown for output weight calculation (b) Single hidden layer feed neural network (SLFN) training computation effort analysis (c) Multiplication analysis for SLFN training ($N = 128, n = 64, m = 30$ and $L = 60$)

rate.

Fig. 3.2(a) measures the timing consumption for the 4 basic linear algebra operations. Clearly, multiplication dominates the computation time. Fig. 3.2(b) shows the time consumption of the four major operations in the training process, which are random input weight generation, multiplication for $preH$, activation and output calculation. In Fig. 3.2(c), matrix-vector multiplication is excluded for output weight calculations. It is clearly shown that more than 64% of time is consumed for matrix-vector multiplication. Therefore, we will design a hardware accelerator with a highly-parallel and pipeline matrix-vector multiplication engine.

3.3 Hardware Implementation

In this section, we will discuss two implementations of neural network training and inference using the direct solver learning algorithm. The first one is a CMOS based implementation using Xilinx FPGA. The second one is a 3D CMOS-RRAM implementation with RRAM based vector-matrix multiplication accelerator.

3.3.1 CMOS based Accelerator

Overview of Computing Flow and Communication

The top level of proposed VLSI architecture for training and inference is shown in Fig. 3.3. The description of this architecture will be introduced based on inference flow. The complex control and data flow of the neural network training and inference are enforced by a top level finite state machine (FSM) with synchronized and customized local module controllers.

For the overall data flow, an asynchronous first-in first-out (FIFO) is designed to

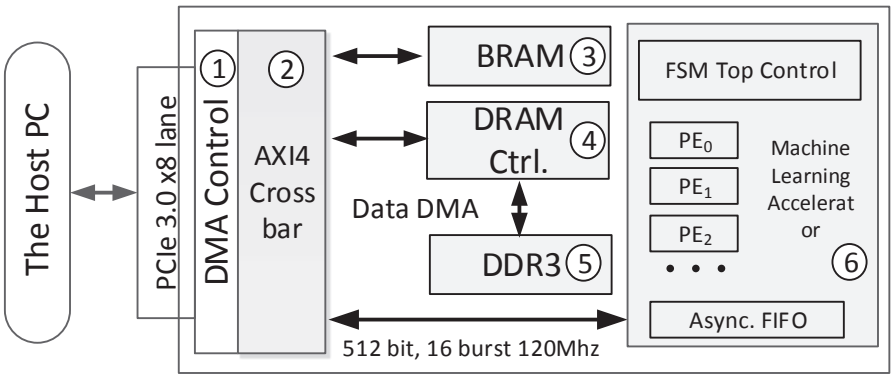


Figure 3.3: Accelerator architecture for training and inference

collect data through AXI4 light from PCIe Gec3X8 [1]. Two buffers are used to store rows of the training data $\boldsymbol{X}$ to perform ping-pong operations. These two buffers will be re-used when collecting the output weight data. To maintain high training accuracy, floating point data is used with parallel fixed point to floating point converter. As the number indicated on each block in Fig. 3.3, data will be firstly collected through PCIe to DRAM and Block RAM (BRAM). BRAM is used to control the core to indicate the read/write address of DRAM during the training/inference process. The core will continuously read data from the BRAM for configurations and starting signal. Once data is ready in DRAM and the start signal is asserted, the core will process computation for neural network inference or training process. An implemented FPGA block design on Vivado is shown in Fig. 3.4.

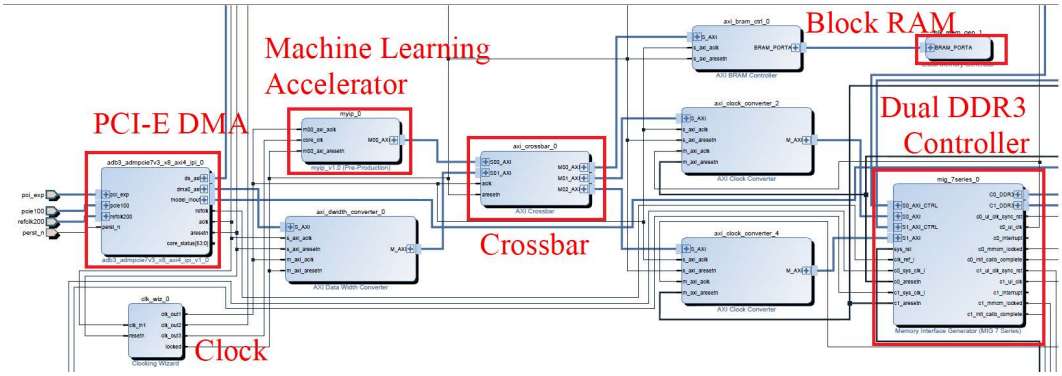


Figure 3.4: Vivado block design for FPGA least-squares machine learning accelerator

A folded architecture is proposed where most modules are re-used among different layers as shown in Fig. 3.5. The *input mux* decides the use of the input data or the output from activation matrix. Input weight can be input from outside or generated inside from linear feedback shift register (LFSR). To achieve similar software-level accuracy,

[1] PCIe is short for Peripheral Component Interconnect Express.

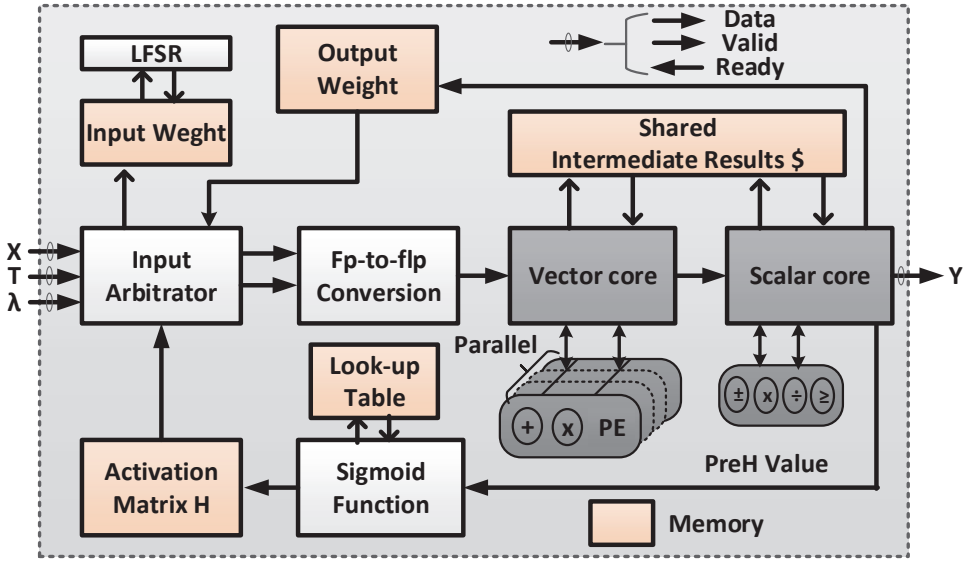


Figure 3.5: Hardware accelerator architecture for online sequential learning

floating-point data is used for both training and inference.

For the neural network inference, the input weight is determined by setting the seed of linear feedback shift register. Random input weight $\boldsymbol{A}$ is generated by padding 126 on exponent positions and 0 on least important mantissa bits to form floating point. The main task of inference is to perform matrix-vector multiplications through vector core (VC) and scalar core (SC) as (3.5). The processing elements (PEs) are configured to perform the matrix-vector computation in parallel and the scalar core is used to accumulate intermediate results. After computing the element in $\boldsymbol{preH}$, a sigmoid function is performed to map the element of $\boldsymbol{preH}$ to $\boldsymbol{H}$ and store in on-chip memory. After matrix $\boldsymbol{H}$ is ready, the output weight is received from the PCIe using the pingpong operations. Again, matrix-vector multiplication is performed and final result is ready to send.

For the neural network training, the first layer computation is the same as inference. However, the computation complexity is high for computing the least-squares solution. Therefore, vector core (VC) and scalar core (SC) are combined to perform complex tasks. Vector core is made of floating point adder and multiplexer, and it is reconfigurable for sum and multiplication. Scalar core is used not only for accumulating PE output results but also for feeding intermediate results back to PE for computation. Vector core and scalar core together can support forward substitution (FS) and backward substitution (BS) for the least-squares problem of training as shown in Step 5 and 8 of Algorithm 2. The data flow control is summarized in Fig. 3.6.

In summary, to improve PE utilization rate, the computing resource for matrix-vector

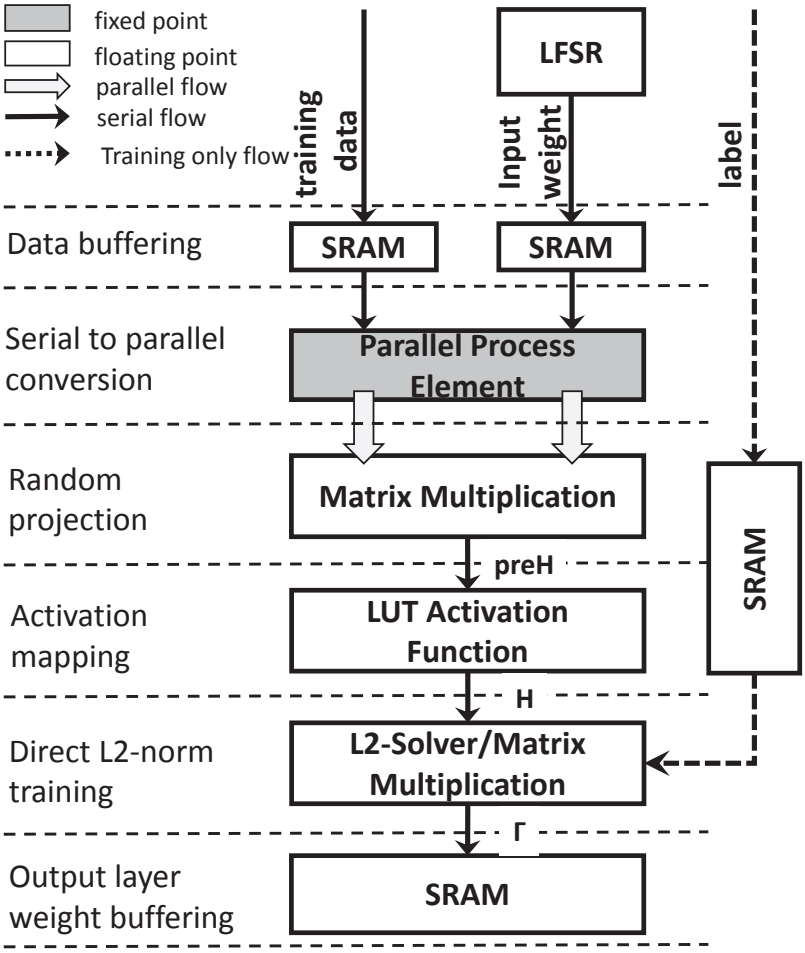


Figure 3.6: Accelerator control and data flow for training

multiplications is shared by the least-squares problem. By exploring the task dependence and resource sharing, we can improve the hardware utilization, allowing more processing elements (PE) to be mapped on FPGA to accelerate the speed.

Least-squares Solver

As mentioned in the reformulated ℓ_2-norm Algorithm 2, Step 5 and 8 require forward substitutions and backward substitutions. Fig. 3.7 provides the detailed mapping on our proposed architecture. For convenience, we use $\boldsymbol{QW} = \boldsymbol{V}$ to represent Step 5 in Algorithm 2, where $\boldsymbol{Q}$ is a triangular matrix. Fig. 3.8 provides the detailed equations in each PE as well as the intermediate values. To explore the maximum level of parallelism, we can perform multiplication at the same time on each row to compute $w_i, i \neq 1$ as shown in the right of Fig. 3.8. However, there are different number of multiplications and accumulations required for different w_i. In the first round, intuitively we require

only 1 PE to perform the multiplication. After knowing w_2, we need 2 parallel PEs for the same computation in the second round. In the end, we need $L-1$ number of PEs to compute. This poses a challenge to perform parallel computation since the number of PE required is increasing. Considering this, we design a shift register, which can store the intermediate results. As such, each PE is independent and can perform the computation continuously by storing results in the shift register. For example, if we have parallelism of 4 for $L = 32$, we can perform 8 times parallel computation and store the results inside registers. This helps improve the flexibility of the process elements (PEs) with better resource utilization.

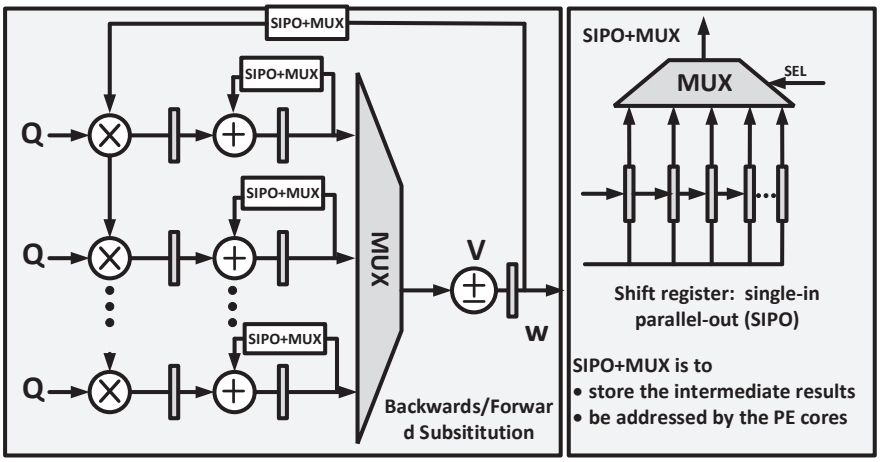


Figure 3.7: Computing diagram of forward/backward substitution in L2-norm solver

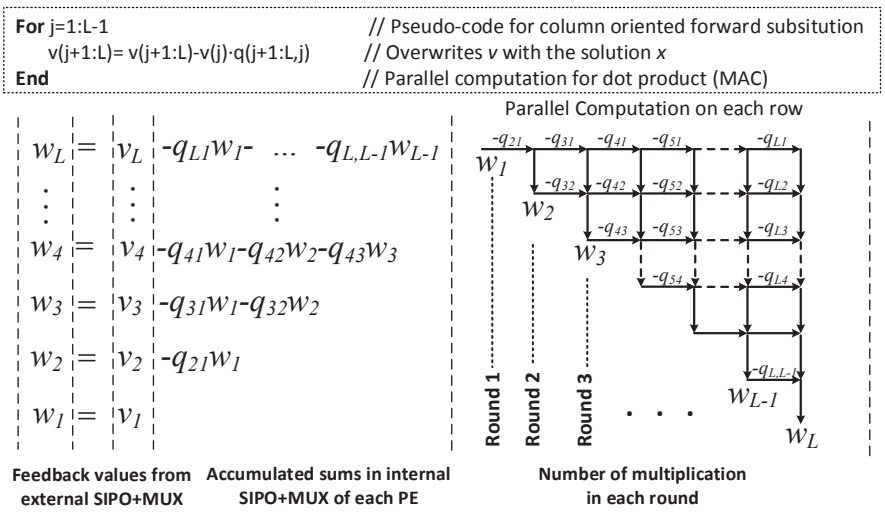


Figure 3.8: Detailed mapping of forward substitution

Matrix-vector Multiplication

All the vector related computation is performed on processing elements (PEs). Our designed PE is similar to [80] but features direct instruction to perform vector-vector multiplications for neural networks. Fig. 3.9 gives an example of vector-vector multiplication (dot product) for (3.5) with parallelism of 4. If the vector length is 8, the folding factor will be 2. The output from PE will be accumulated twice based on the folding factor before sending out the vector-vector multiplication result. The adder tree will be generated based on the parallelism inside the vector core. The output will be passed to scalar core for accumulations. In the PE, there is a bus interface controller. It will control the multiplicand of PE and pass the correct data based on the top control to PE. The multiplier can be bypassed by setting one of the multiplicand 1 and the multiplication result will be sent through bus interface controller to process summation operation. The adding process in PE can also be bypassed by sending 0 to its input.

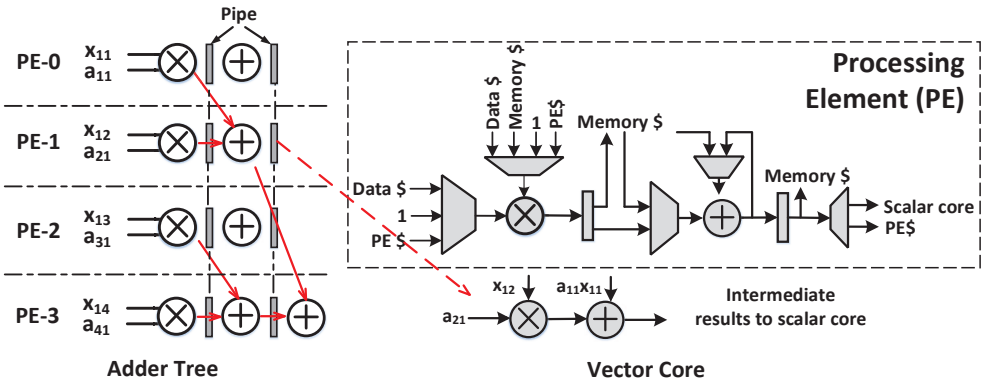


Figure 3.9: Computing diagram of matrix-vector multiplication

LUT based Sigmoid Function

The Sigmoid function is used in (3.5) and defined as $f(x) = \frac{1}{1+e^{-x}}$. The simplest implementation is to use look-up table. To save silicon area, we make use of the odd symmetric property of Sigmoid function. The non-saturation range is between -8 and 8. Any value beyond the saturation region will be set to 0 and 1 respectively. To further reduce the memory required, we divide the non-saturation region into more small regions based on the exponent of 2. The quantization steps will get smaller when the input closes to 0, which is the linear region. Around $4K$ bits of memory is used for Sigmoid function implementation with a maximum error less than 0.01.

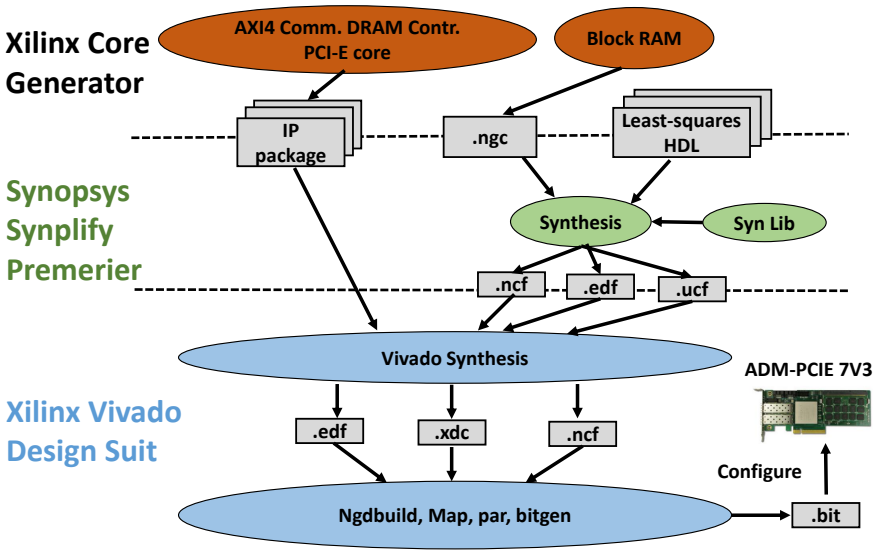


Figure 3.10: CAD flows for implementing least-squares on ADM-PCIe 7V3

FPGA Design Platform and CAD Flow

The ADM-PCIE-7V3 is a high-performance reconfigurable computing card intended for high speed performance applications, featuring a Xilinx Virtex-7 FPGA. The key features of ADM-PCIE 7V3 can refer to [89].

The development platform is mainly on Vivado 14.4. The direct memory access (DMA) bandwidth is 4.5GB/s. On the FPGA board ADM-PCIEe7V3, the DDR3 bandwidth is 1333MT/s with 64 bits width. The CAD flow for implementing the machine learning accelerator on the ADM-PCIe 7V3 [89] is shown in Fig. 3.10. The Xilinx CORE Generator System is first used to generate the data memory macros that are mapped to the Block RAM (BRAM) resources on the FPGA. The generated NGC files contain the design netlist, constraints files and Verilog wrapper. Then, these files together with the RTL codes of the machine learning accelerator are loaded to Synplify Premier for logic synthesis. Note that the floating-point arithmetic units used in our design are from the Synopsys DesignWare library. The block RAM is denoted as a black box for Synplify synthesis. The gate-level netlist is stored in the electronic data interchange format (EDIF), and the user constraint file (UCF) contains user-defined design constraints. Next, the generated files are passed to Xilinx Vivado Design Suite to merge with other IP cores such as DRAM controllers and PCIe cores. In the Vivado design environment, each IP is packaged and connected. Then, we synthesize the whole design again under Vivado environment. Specifically, the *ngbbuild* command reads the netlist in EDIF format and creates a native generic database (NGD) file. This NGD file contains a logical description of the design and a description of the original design hierarchy. The

map command takes the NGD file, maps the logic design to the specific Xilinx FPGA, and outputs the results to a native circuit description (NCD) file. The *par* command takes the NCD file, places and routes the design, and produces a new NCD file, which is then used by the *bitgen* command to generate the bit file for FPGA programming.

3.3.2 RRAM-crossbar based Accelerator

In this section, RRAM device and RRAM-crossbar based in-memory computation are introduced first. Thereafter, we will illustrate the detailed 3D multi-layer CMOS-RRAM accelerator for machine learning on neural network. Furthermore, we will illustrate how to map matrix-vector multiplication on RRAM-crossbar layer, which has three steps: parallel digitalization, XOR and encoding. These steps are implemented in layer 2. In addition, CMOS-ASIC accelerator is also designed in layer 3 for the remaining operations such as division and non-linear mapping in a pipelined and parallel fashion. The detailed mapping of these three steps can be referred to [90].

RRAM-Crossbar Device

Emerging resistive random access memory (RRAM) [91, 92] is a two-terminal device with 2 non-volatile states: high resistance state (HRS) and low resistance state (LRS). As RRAM states are sensible to the input voltages, special care needs to be taken while reading, such that the read voltage V_r is less than half of write voltage V_w. The read voltage V_r and the write voltage V_w are related as follows

$$V_w > V_{th} > V_w/2 > V_r, \tag{3.19}$$

where V_{th} is the threshold voltage of RRAM.

In one RRAM-crossbar, given the input probing voltage, the current on each bit-line (BL) is the multiplication-accumulation of current through each RRAM device on the BL. Therefore, the RRAM-crossbar array can intrinsically perform the analog matrix-vector multiplication [93]. Given an input voltage vector $V_{WL} \in \mathbb{R}^{M\times 1}$, the output voltage vector $V_{BL} \in \mathbb{R}^{N\times 1}$ can be expressed as

$$\begin{bmatrix} V_{BL,1} \\ \vdots \\ V_{BL,N} \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,M} \\ \vdots & \ddots & \vdots \\ c_{N,1} & \cdots & c_{N,M} \end{bmatrix} \begin{bmatrix} V_{WL,1} \\ \vdots \\ V_{WL,M} \end{bmatrix} \tag{3.20}$$

where $c_{i,j}$ is the configurable conductance of the RRAM resistance $R_{i,j}$, which can represent a real-value weight. Compared to traditional CMOS implementation, RRAM

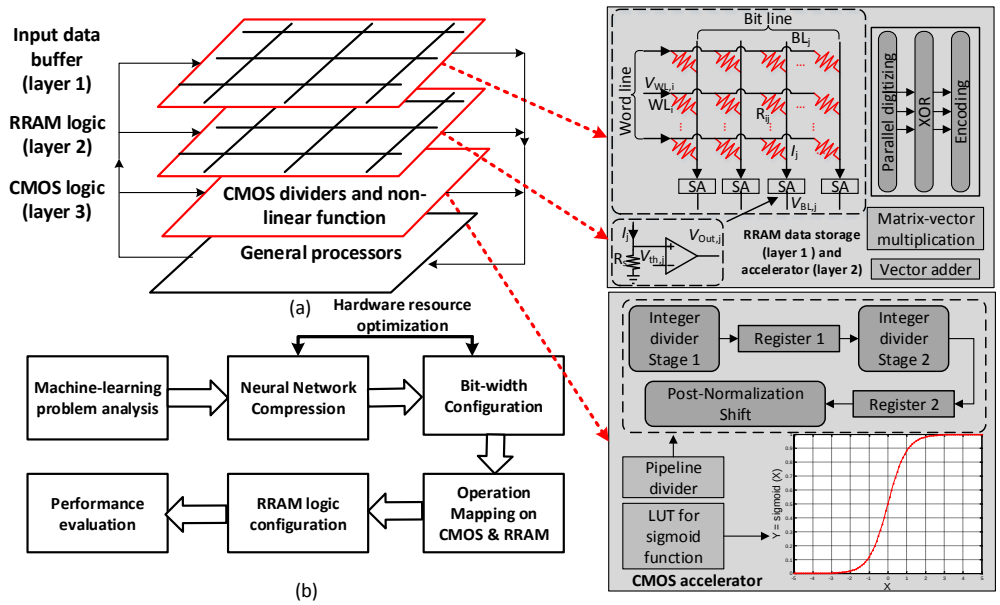


Figure 3.11: (a) 3D multi-layer CMOS-RRAM accelerator architecture; (b) Machine learning algorithm mapping flow on proposed accelerator

crossbar achieves better parallelism and consumes less power. However, note that analog implementation of matrix-vector multiplication is strongly affected by non-uniform resistance values [94]. As such, one needs to develop a digital fashioned multiplication based on the RRAM-crossbar instead. Therefore, a digital-fashioned multiplication on RRAM-crossbar is preferred to minimize the device non-uniform impact from process variation [90].

3D Multi-layer CMOS-RRAM Architecture

Recent work [95] has shown that the 3D integration supports heterogeneous stacking because different types of components can be fabricated separately, and layers can be stacked and implemented with different technologies. Therefore, stacking non-volatile memories on top of microprocessors enables cost-effective heterogeneous integration. Furthermore, works in [96, 97] have also shown the feasibility to stack RRAM on CMOS to achieve smaller area and lower energy consumption.

The proposed 3D multi-layer CMOS-RRAM accelerator with three layers is shown in Fig. 3.11(a). This accelerator is composed of a two-layer RRAM-crossbar and a one-layer CMOS circuit. As Fig. 3.11(a) shows, layer 1 of RRAM-crossbar is implemented as a buffer to temporarily store input data to be processed. Layer 2 of RRAM-crossbar performs logic operations such as matrix-vector multiplication and also vector addition. Note that buffers are designed to separate resistive networks between layer 1 and layer

2. The last layer of CMOS contains read-out circuits for RRAM-crossbar and performs as logic accelerators designed for other operations besides matrix-vector multiplication, including pipelined divider, look-up table (LUT) designed for division operation and activation function in machine learning.

Moreover, Fig. 3.11(b) shows the working flow for machine learning mapped to the proposed architecture. Firstly, the detailed architecture of machine learning (ML) (e.g. number of layers and activation function) is determined based on the accuracy requirements and data characteristics. Secondly, operations of this machine learning algorithm are analyzed and reformulated so that all the operations can be accelerated in 3D multi-layer CMOS-RRAM architecture as illustrated in Fig. 3.11(a). Furthermore, the bit-width operating on RRAM-crossbar is also determined by balancing the accuracy loss and energy saving. Finally, logic operations on RRAM-crossbar and CMOS are configured based on the reformulated operations, energy saving and speed-up.

Such a 3D multi-layer CMOS-RRAM architecture has advantages in three manifolds. Firstly, by utilizing RRAM-crossbar for input data storage, leakage power of memory is largely removed. In a 3D architecture with TSV interconnection, the bandwidth from this layer to next layer is sufficiently large to perform parallel computation. Secondly, RRAM-crossbar can be configured as computational units for the matrix-vector multiplication with high parallelism and low power. Lastly, with an additional layer of CMOS-ASIC, more complicated tasks such as division and non-linear mapping can be performed. As a result, the whole training process of machine learning can be fully mapped to the proposed 3D multi-layer CMOS-RRAM accelerator architecture towards real-time training and inference.

Data Quantization

To implement the whole training algorithm on the proposed 3D multi-layer CMOS-RRAM accelerator, the precision of the real values requires a careful evaluation. Compared to the software double precision floating point format (64-bit), the values in the training algorithm are truncated into finite precision. A general N_b bit fixed point representation is shown as follows.

$$y = -b_{N_b-1}2^{m_b} + \sum_{l=0}^{N_b-2} b_l 2^{l-n_b} \tag{3.21}$$

where m_b represents bit-width for the integer and n_b represents bit-width for the fraction point.

In the proposed accelerator, we first assign a fixed 16 bit-width for normalized input

data such as $\boldsymbol{X}$,$\boldsymbol{Y}$ and $\boldsymbol{D}$ with scale factor $m_b = 0$ and $n_b = 16$. For weights such as $\boldsymbol{A}$ and $\boldsymbol{B}$ in single layer feed forward neural network (SLFN), we determine m_b by finding the logarithm of dynamic range (i.e. $log_2(Max - Min)$) and n_b is set as $16 - m_b$. Furthermore, we applied greedy search method based on the inference accuracy to find the optimal bit width. The bit width is reduced by cross-validation applied to evaluate the effect. By dynamic tuning the bit width, our objective is to find the minimum bit width with acceptable inference accuracy loss.

RRAM Layer Implementation for Digitized Matrix-vector Multiplication

The matrix-vector multiplication in the neural network is explained using $y_{ij} = \sum_{k=1}^{L} h_{ik}\gamma_{kj}$ as an example. Such multiplication can be expressed in binary multiplication [90].

$$\begin{aligned} y_{ij} &= \sum_{k=1}^{L}(\sum_{e=0}^{E-1} B_e^{h_{ik}} 2^e)(\sum_{g=0}^{G-1} B_g^{\gamma_{kj}} 2^g), \\ &= \sum_{e=0}^{E-1}\sum_{g=0}^{G-1}(\sum_{k=1}^{L} B_e^{h_{ik}} B_g^{\gamma_{kj}} 2^{e+g}) = \sum_{e=0}^{E-1}\sum_{g=0}^{G-1} s_{eg} 2^{e+g} \end{aligned} \tag{3.22}$$

where s_{eg} is the accelerated result from RRAM-crossbar. $B^{h_{ik}}$ is the binary bit of h_{ik} with E bit-width and $B^{\gamma_{kj}}$ is the binary bit of γ_{kj} with G bit-width. As mentioned above, bit-width E and G are decided using cross validation in design to achieve balanced accuracy and hardware usage [94].

Step 1: Parallel Digitalization: It requires an $L \times L$ RRAM-crossbar. Each inner-product is produced by the RRAM-crossbar as shown in Fig. 3.12(a). h is set as cross-bar input and γ is written in RRAM cells. In the $L \times L$ RRAM-crossbar, resistance of RRAMs in each column are the same, but V_{th} among columns are different. As a result, the output of each column mainly depends on ladder-like threshold voltages $V_{th,j}$. If the inner-product result is s, the output of step 1 is like $(1...1,0...0)$, where $O_{1,s} = 1$ and $O_{1,s+1} = 0$.

Step 2: XOR: It is to identify the index of s with the operation $O_{1,s} \oplus O_{1,s+1}$. Note that $O_{1,s} \oplus O_{1,s+1} = 1$ only when $O_{1,j} = 1$ and $O_{1,j+1} = 0$ from Step 1. The mapping of RRAM-crossbar input and resistance is shown in Fig. 3.12(b), and threshold voltage configuration is $V_{th,j} = \frac{V_r R_s}{2R_{on}}$. Therefore, the index of s is identified by XOR operation.

Step 3: Encoding: the third step produces s in the binary format as an encoder with the thresholds from Step 2. The input from the second step produces $(0...1,0...0)$ like result where only the s-th input is 1. As a result, only the s-th row is read out and no current merge occurs in this step. The corresponding binary format $binary(s)$ is an intermediate result and stored in the s-th row, as shown in Fig. 3.12(c). Encoding step

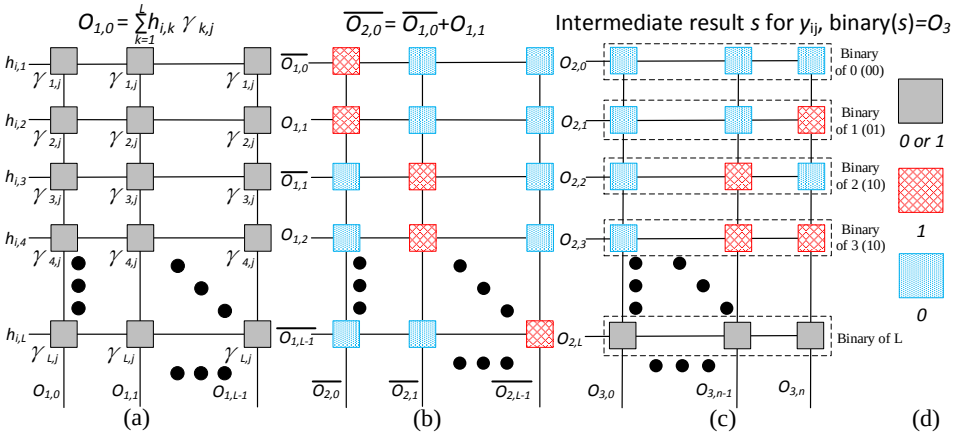


Figure 3.12: Detailed mapping for digitalized matrix-vector multiplication on RRAM-crossbar: (a) Parallel digitalization (b) XOR (c) Encoding and (d) Legend of mapping

needs an $L \times n$ RRAM-crossbar, where $n = \lceil \log_2 L \rceil$ is the number of bits in order to represent 1 to L in binary format.

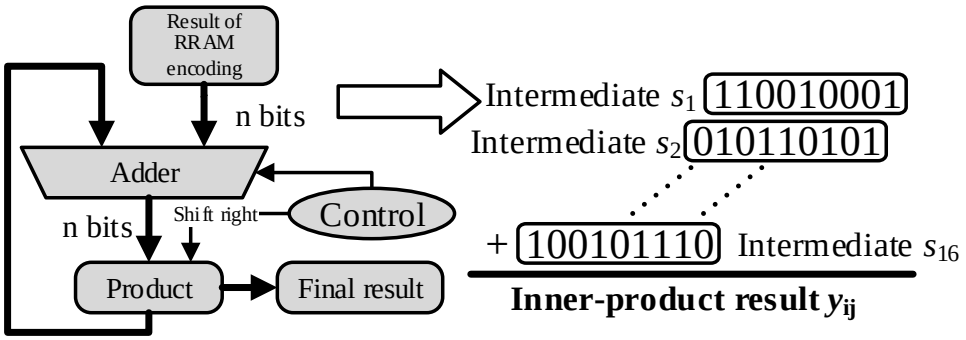


Figure 3.13: Inner-product operation on RRAM-crossbar

CMOS Layer Implementation for Decoding and Incremental Least-squares In the CMOS layer, decoding and more complex operations for incremental least-squares solution are designed. Since the output from RRAM layer is in the binary format, decoding is required to obtain real values. Adder and shifter are designed in layer 3 with CMOS to complete this process as shown in Fig. 3.13.

To fully map the incremental machine learning on the proposed 3D multi-layer CMOS-RRAM accelerator, the following operations are needed in CMOS-ASIC including: sorting (3.2), non-linear mapping (3.5) and division (3.18). Fig. 3.5 shows the detailed mapping of the supervised classification on the proposed 3D multi-layer CMOS-RRAM accelerator. In this case, the RRAM logic layer (layer 2) will work as vector cores with parallel processing elements (PE) for multiplication and summation.

The CMOS scalar core is implemented to perform scalar operation including division. A sequential divider is implemented with 5-stage pipelines to reduce critical path latency. Non-linear mapping such as Sigmoid function for activation is also implemented in look-up table (LUT) (3.5). As a result, the whole training process can be mapped to the proposed 3D multi-layer CMOS-RRAM accelerator with small accuracy loss.

3.4 Experiment Results

3.4.1 CMOS based results

In this section, we firstly discuss the experiment set-up and benchmarks for comparisons. The performance of proposed scalable architecture is evaluated for regression problem and classification problem respectively. Finally, the energy consumption and speed-up of proposed accelerator are evaluated in comparison with CPU, embedded CPU and GPU. The proposed CMOS accelerator will be applied in Chapter 5 for IoT applications.

Experiment Set-up and Benchmarks

To verify our proposed architecture, we have implemented it on Xilinx Virtex 7 with PCI Express Gen3x8 [89]. The HDL code is synthesized using Synplify and the maximum operating frequency of the system is 53.1 MHz under 128 parallel PEs. The critical path is identified as the floating-point division, where 9 stages of pipeline are inserted to speed-up. We develop three baselines (x86 CPU, ARM CPU and GPU) for performance comparisons.

Baseline 1: General Processing Unit (x86 CPU). The general CPU implementation is based on C program on a computer server with Intel Core -i5 *3.20GHz* core and *8.0GB* RAM.

Baseline 2: Embedded Processor (ARM CPU) . The embedded CPU (Beagle-Board-xM) [98] is equipped with 1GHz ARM core and 512MB RAM. The implementation is performed using C program under Ubuntu 14.04 system.

Baseline 3: Graphics Processing Unit (GPU). The GPU implementation is performed by CUDA C program with cuBLAS library. A Nvidia GeForce GTX 970 is used for the acceleration of learning on neural network.

The testing environment of Alpha Data 7V3 is shown as Fig. 3.14.

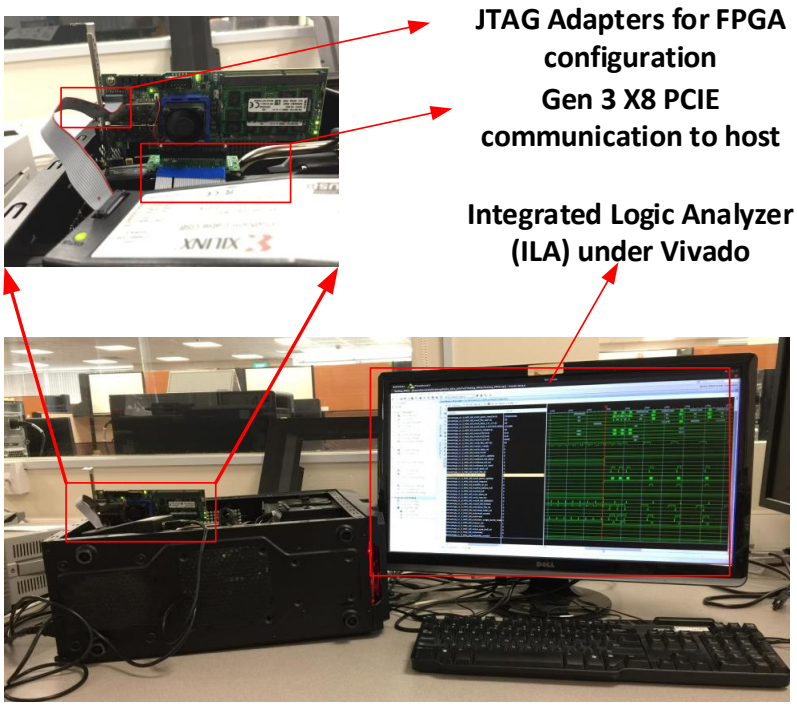


Figure 3.14: Inference enviroment on the ADM-PCIE 7V3 evaluation platform

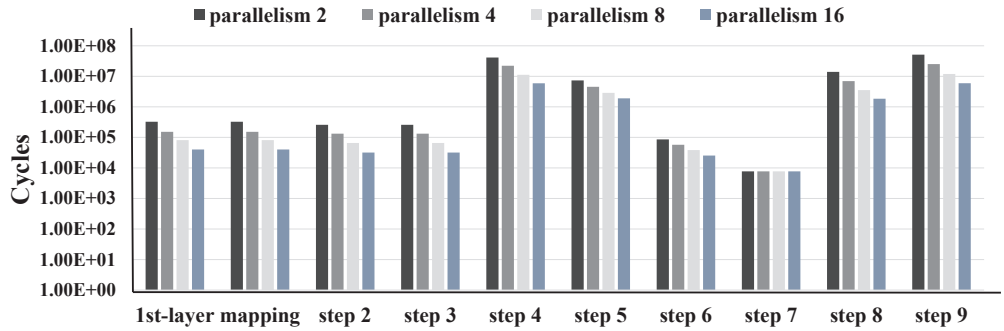


Figure 3.15: Training cycles at each step of the proposed training algorithm with different parallelisms ($N = 74$; $L = 38$; $M = 3$ and $n = 16$)

Scalable and Parameterized Accelerator Architecture

The proposed accelerator architecture features great scalability for different applications. Table 3.2 shows all the user-defined parameters supported in our architecture. At circuit level, users can adjust the stage of pipeline of each arithmetic to satisfy the speed, area and resource requirements. At architecture level, the parallelism of PEs can be specified based on the hardware resource and speed requirements. The neural network parameters n, N, H can also be reconfigured for specific applications.

Fig. 3.15 shows the training cycles on each step on proposed training algorithms for synthesized dataset. Different parallelisms P are applied to show the speed-up of

Table 3.2: Tunable parameters on proposed architecture

Parameters		Descriptions
Circuits	{MAN EXP}	Word-length of mantissa, exponent
	$\{P_A, P_M, P_D, P_C\}$	Pipe. stages of adder, mult, , div and comp
Architectures	P	Parallelism of PE in VC
	n	Maximum signal dimensions
	N	Maximum training/inference data size
	H	Maximum number of hidden nodes

Table 3.3: Resource utilization under different parallelism levels ($N = 512$, $H = 1024$, $n = 512$ and $50Mhz$ clock)

Paral.	LUT	Block RAM	DSP
8	52614 (12%)	516 (35%)	51 (1.42%)
16	64375 (14 %)	516 (35%)	65 (1.81%)
32	89320 (20 %)	516 (35%)	96 (2.67%)
64	139278 (32 %)	516 (35%)	160 (4.44%)
128	236092 (54 %)	516 (35%)	288 (8.00%)

each steps. The speed-up of 1st-layer for matrix-vector multiplication is scaling up with the parallelism. The same speed-up improvement is also observed in step 3, 4 and 9 in Algorithm 2, where the matrix-vector multiplication is the dominant operation.

However, when the major operation is the division for the backward and forward substitution, the speed-up is not that significant and tends to saturate when the division becomes the bottleneck. We can also observe in Step 7, the memory operations do not scale with parallelism. It clearly shows that matrix-vector multiplication is the dominant operation in the training procedure (1st Layer, step 3, step 4 and step 9) and our proposed accelerator architecture is scalable to dynamically increase the parallelism to adjust the speed-up.

The resource utilization under different parallelisms is achieved from Xilinx Vivado after placement and routing. From Table 3.3, we can observe that LUT and DSP are almost linearly increasing with parallelism. However, block RAM (BRAM) keeps constant with increasing parallelisms. This is because BRAM is used for data buffer, which is determined by other architecture parameters (N,H,n). Fig. 3.16 shows the layout from Vivado of the designed least-squares solver.

Performance for Data Classification

In this experiment, six datasets are trained and tested from UCI dataset [99], which are wine, car, dermatology, zoo, musk and Connectionist Bench (Sonar, Mines vs. Rocks).

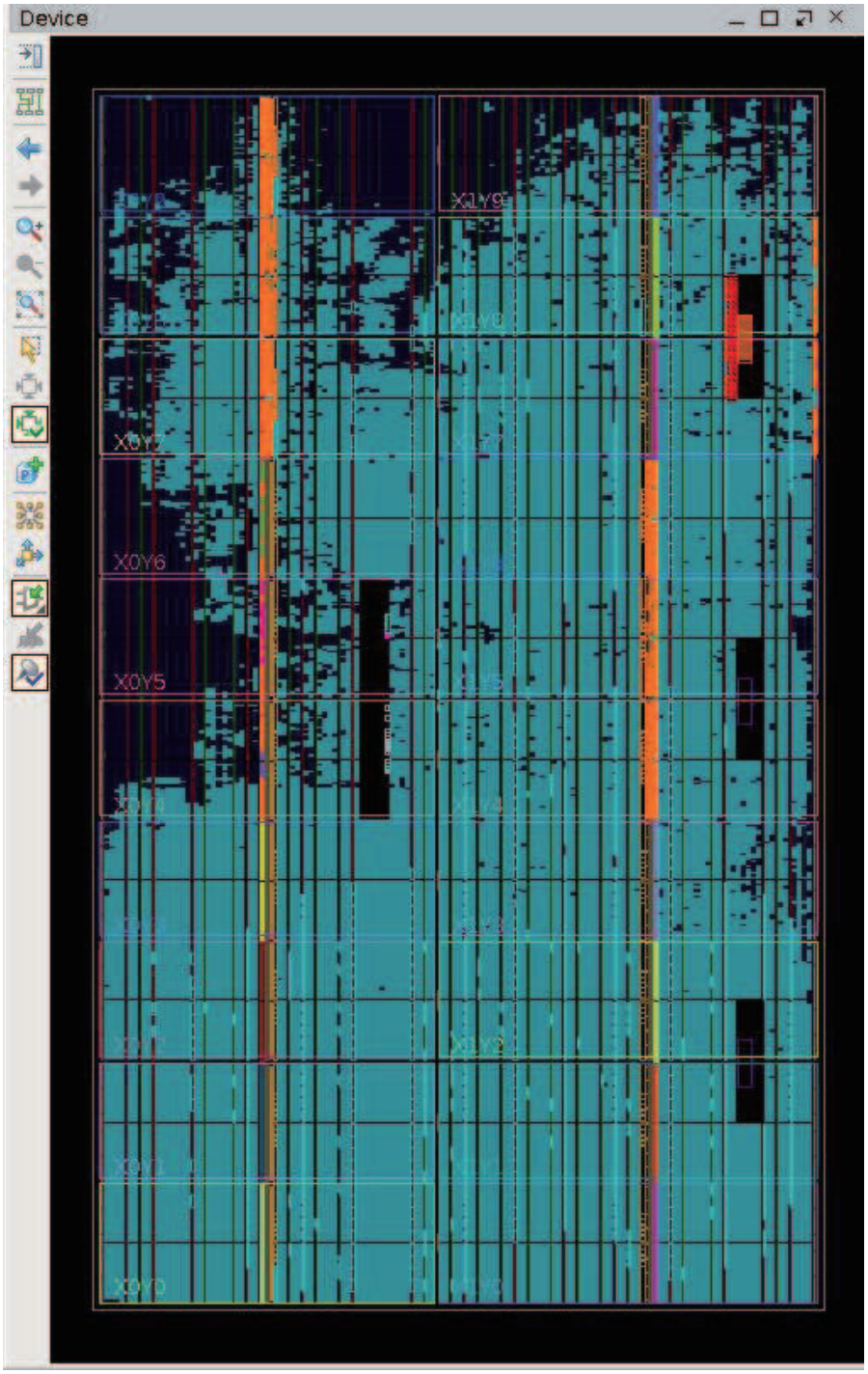


Figure 3.16: Layout view of the FPGA with least-squares machine learning accelerator implemented

Table 3.4: UCI Dataset Specification and Accuracy

Benchmarks	Data Size	Dim.	Class	Node No.	Acc.
Car	1728	6	4	256	90.90%
Wine	178	13	3	1024	93.20%
Dermatology	366	34	6	256	85.80%
Zoo	101	16	7	256	90.00%
Musk1	476	166	2	256	69.70%
Conn.Bench	208	60	2	256	70%

The details of each dataset are summarized in Table 3.4. The UCI dataset is developed for performance comparisons under different benchmarks. The proposed technique will be adopted in Chapter 5 for IoT applications. The architecture is set according to the training data set size and dimensions to demonstrate the parameterized architecture. For example, $N = 128$ represents that the architecture parameter (maximum training size) is 128 with the actual dataset wine size of 74. The accuracy result is summarized in Table 3.4. The accuracy of FPGA is the same comparing to Matlab result since the single floating-point data format is applied to the proposed architecture.

For the speed-up comparison, our architecture will not only compare to the time consumed by least-squares solver (direct-solver) training method, but also SVM [100] and backwards propagation (BP) based method [77] on CPU. For connectionist bench dataset, the speed-up of proposed accelerator is as high as 24.86×, when compared to the least-squares solver software solution on CPU. Furthermore, 801.20× and 25.55× speed-up can be achieved comparing to BP and SVM on CPU. Table 3.5 provides detailed speed-up comparisons of each dataset with 32 parallel processing elements. Note that on the same CPU platform, the direct solver (DS) is also around 100× faster than backwards propagation (BP) for the car dataset. Similar performance improvement is also observed for other datasets.

Performance for Data Regression

For the regression problem, we use the short-term load forecasting as an example to demonstrate it. The dataset for residential load forecasting is collected by Singapore Energy Research Institute (ERIAN). The dataset consists of 24-hour energy consumption, occupants motion and environmental records such as humidity and temperatures from 2011 to 2015. We will perform multi-hours ahead load forecasting using real-time environmental data, occupants motion data and previous hours and days energy consumption data. Model will be retrained sequentially after new training data is generated. We will come back to this application for energy management system in Chapter 5.4.

To quantize the load forecasting performance, we use two metrics: root mean square

Table 3.5: Performance comparisons between direct solver (DS) and other solvers on FPGA and CPU

Benchmarks	DS (FPGA)	DS (CPU)	BP (CPU)	SVM (CPU)	Imp. (CPU)
Car	44.3 ms	370 ms	36980 ms	1182 ms	8.35×
Wine	207.12 ms	360 ms	11240 ms	390 ms	1.74×
Dermatology	19.45 ms	160 ms	17450 ms	400 ms	8.23×
Zoo	22.21 ms	360 ms	5970 ms	400 ms	16.21×
Musk	24.09 ms	180 ms	340690 ms	3113 ms	7.47×
Conn.Bench	14.48 ms	360 ms	11630 ms	371 ms	24.86×

Table 3.6: Load forecasting accuracy comparison and accuracy improvement comparing to FPGA results to SVM result

Machine Learning	MAPE			RMSE		
	Max	Min	Avg	Max	Min	Avg
NN FPGA	0.12	0.072	0.92	18.87	6.57	12.80
NN CPU	0.11	0.061	0.084	17.77	8.05	12.85
SVM	0.198	0.092	0.135	20.91	5.79	15.13
Imp. (CPU)(%)	4.28	-18.03	-9.52	-6.19	18.39	0.39
Imp. (SVM)(%)	41.01	21.74	31.85	9.76	-13.47	15.40

error (RMSE) and mean absolute percentage error (MAPE). Table 3.6 is the summarized performance with comparison of SVM. We can observe that the our proposed accelerator has almost the same performance as CPU implementation. It also shows an average of 31.85% and 15.4% improvement for MAPE and RMSE comparing to SVM based load forecasting.

Performance Comparisons with other Platforms

In the experiment, the maximum throughput of proposed architecture is 12.68$Gflops$ with 128 parallelism for matrix multiplication. This is slower than 59.78$Gflops$ GPU based implementation but higher than 5.38$Gflops$ x86 CPU based implementation.

To evaluate the energy consumption, we calculate the energy for a given implementation by multiplying the peak power consumption of corresponding devices. Although this is a pessimistic analysis, it is still very likely to reach due to intensive memory and computation operations. Table 3.7 provides detailed comparisons between different platforms. Our proposed accelerator on FPGA has the lowest power consumption (0.85W) comparing to GPU implementation (145W), ARM CPU (2.5W) and x86 CPU implementation (84W). For the training process, although GPU is the fastest implementation, our accelerator still has 2.59× and 51.22× speed-up for training comparing to x86 CPU and ARM CPU implementations. Furthermore, our proposed method shows 256.0×, 150.7× and 2.95× energy saving comparing to CPU, ARM CPU and GPU respectively

Table 3.7: Proposed architecture performance in comparison with other computation platform

Platform	Type	Format	Time	Power	Energy	Speed-up	E. Imp.
x86 CPU	Train	Single	1646ms	84W	138.26J	2.59X	256.0X
	Inference		1.54ms	84W	0.129J	4.56X	450.2X
ARM CPU	Train	Single	32550ms	2.5W	81.38J	51.22X	150,7X
	Inference		30.1ms	2.5W	0.0753J	89.05X	261.9X
GPU	Train	Single	10.99ms	145W	1.594J	0.017X	2.95X
	Inference		0.196ms	145W	0.0284J	0.580X	98.92X
FPGA	Train	Single+	635.4ms	0.85W	0.540J	–	–
	Inference	Fixed	0.338ms	0.85W	0.287mj	–	–

for training model. For the inference process, it is mainly on matrix-vector multiplications. Therefore, GPU based implementation provides better speed-up performance. However, our proposed method still has 4.56× and 89.05× speed-up for inference comparing to x86 CPU and ARM CPU implementations respectively. Moreover, our accelerator is the most low-power platform with 450.1×, 261.9× and 98.92× energy saving comparing to x86 CPU, ARM CPU and GPU based implementations respectively.

3.4.2 RRAM based results

Experiment Set-up and Benchmarks

In the experiment, we have implemented three baselines for performance comparisons. The detail of each baseline is listed below:

Baseline 1: General Processor. The general processor implementation is based on Matlab on a computer server with 3.46GHz core and 64.0GB RAM.

Baseline 2: Graphics Processing Unit (GPU). The GPU implementation is performed by Matlab GPU parallel toolbox on the same server. A Nvidia GeForce GTX 970 is used for the acceleration of matrix-vector multiplication operations for learning on neural network.

Baseline 3: 3D-CMOS-ASIC. The 3D-CMOS-ASIC implementation with proposed architecture is done by Verilog with 1GHz working frequency based on CMOS 65nm low power PDK. Power, area and frequency are evaluated through Synopsys DC compiler (D-2010.03-SP2). Through-silicon via (TSV) model in [101] is included for area and power evaluation. 512 vertical TSVs are assumed between layers to support communication and parallel computations [102].

Proposed 3D-RRAM-CMOS: The settings of CMOS evaluation and TSV model are the same as baseline 3. For the RRAM-crossbar design evaluation, the resistance of RRAM is set as 1K and 1M as on-state and off-state resistance respectively according to

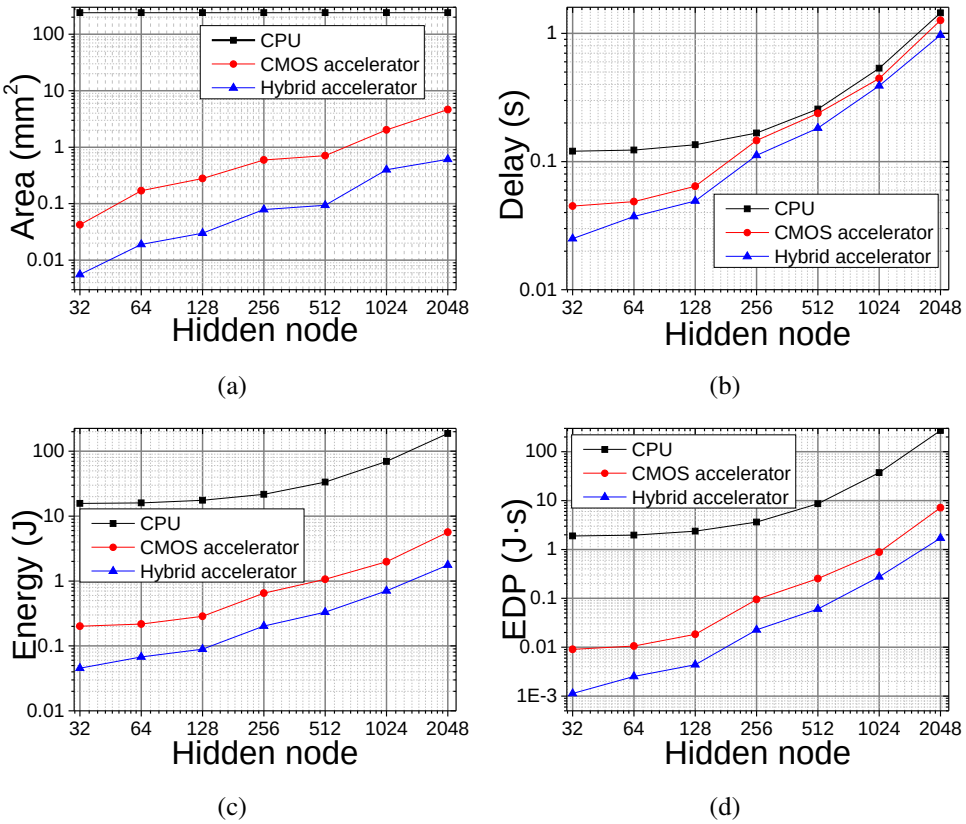


Figure 3.17: Scalability study of hardware performance with different hidden node numbers for: (a) area; (b) delay; (c) energy and (d) energy-delay-product

[103] with working frequency of 200MHz.

3D Multi-layer CMOS-RRAM Accelerator Scalability Analysis

To evaluate the proposed 3D multi-layer CMOS-RRAM architecture, we perform the scalability analysis of energy, delay and area on glass UCI dataset [99]. In SLFN, the number of hidden nodes may change depending on the accuracy requirement. As a result, the improvement of proposed accelerator with different L from 64 to 256 is evaluated as shown in Fig. 3.17. With increasing L, more computing units are designed in 3D-CMOS-ASIC and RRAM-crossbar to evaluate the performance. When L reaches 256, RRAM-crossbar can achieve 2.1× area-saving and 10.02× energy-saving compared to 3D-CMOS-ASIC. In Fig. 3.17(d), energy-delay-product (EDP) of RRAM-crossbar increases faster than 3D-CMOS-ASIC. As the size of RRAM-crossbar is proportional to the square of L, the EDP improvement of RRAM-crossbar is less with larger L. However, it still shows great advantage in EDP with 51× better than 3D-CMOS-ASIC when L is 500, which is a large number of hidden nodes for glass benchmark of 9 features.

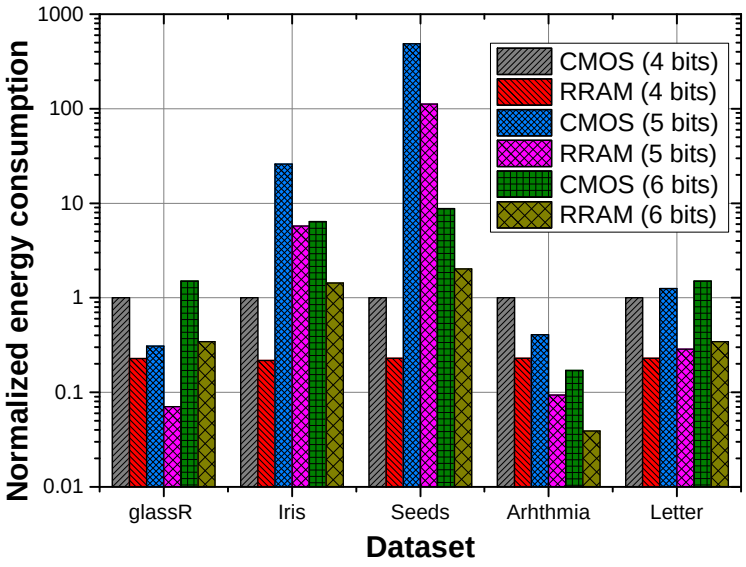


Figure 3.18: Energy-saving comparison under different bit-width of CMOS and RRAM with the same accuracy requirement

3D Multi-layer CMOS-RRAM Accelerator Bit-width Configuration Analysis

Table 3.8 shows the inference accuracy under different datasets [99, 104] and configurations of support vector machine (SVM) and single layer feed-forward neural network (SLFN). It shows that accuracy of classification is not very sensitive to the RRAM bit-width configuration. For example, the accuracy of Iris dataset can work with negligible accuracy loss at 5 RRAM bit-width. When the RRAM bit-width increased to 6, it performs the same as 32 bit-width configurations. Similar observation is found in [94] by truncating algorithms with limited precision for better energy efficiency. Please note that training data and weight related parameters are quantized to perform matrix-vector multiplication on RRAM crossbar accelerator.

Fig. 3.18 shows the energy comparisons under different bit-width configurations for CMOS and RRAM under the same accuracy requirements. An average of 4.5$\times$ energy saving can be achieved for the same number of bit-width configuration. The energy consumption is normalized by the CMOS 4 bit-width configuration. Furthermore, we can observe that not smaller number of bits always achieves better energy saving. Fewer number of bit-width may require much larger neural network to achieve the required classification accuracy. As a result, its energy consumption increases.

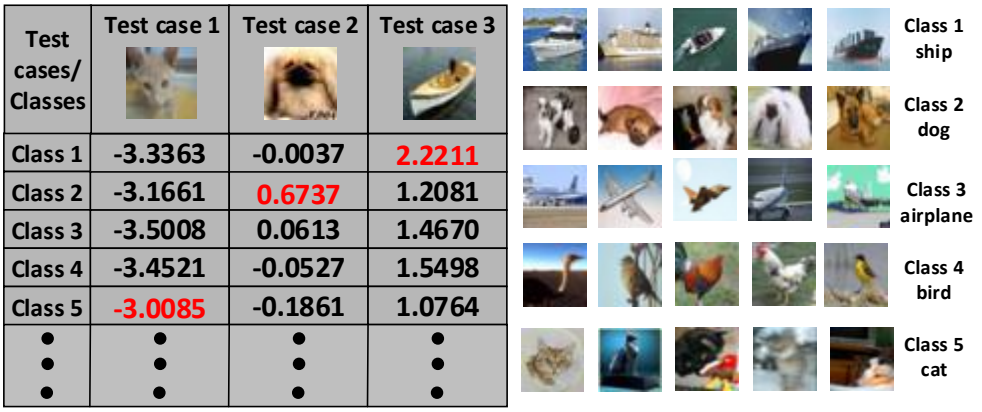


Figure 3.19: On-line machine-learning for image recognition on the proposed 3D multi-layer CMOS-RRAM accelerator using benchmark CIFAR-10

Table 3.8: Inference accuracy of ML techniques under different dataset and configurations (normalized to all 32 bits)

Datasets	Size	Feat.	Cl.	4 bit Acc.(%)		5 bit Acc.(%)		6 bit Acc.(%)	
				SVM‡	&SLFN	SVM‡	&SLFN	SVM‡	&SLFN
Glass	214	9	6	100.07	100.00	93.88	99.30	99.82	99.30
Iris	150	4	3	98.44	94.12	100.00	94.18	100.00	100.00
Seeds	210	7	3	97.98	82.59	99.00	91.05	99.00	98.51
Arrhythmia	179	13	3	96.77	97.67	99.18	98.83	99.24	100.00
Letter	20,000	16	7	97.26	53.28	98.29	89.73	99.55	96.13
CIFAR-10	60,000	1600†	10	98.75	95.71	99.31	97.06	99.31	99.33

† 1600 features extracted from 60,000 32×32 color images with 10 classes. ‡ Least-square SVM is used for comparison.

3D Multi-layer CMOS-RRAM Accelerator Performance Analysis

Fig. 3.19 shows the classification values on image data [104] with an example of 5 classes. As the discussion of (3.2), the index with maximum values (highlighted in red) is selected to indicate the class of the test case. A few sample images are selected. Please note that 50,000 and 10,000 images are used for training and inference respectively.

In Table 3.9, performance comparisons among Matlab, 3D-CMOS-ASIC and 3D multi-layer CMOS-RRAM accelerator are presented, and the acceleration of each procedure based on the formula described in Chapter 3.2.2 is also addressed. Among the three implementations, 3D multi-layer CMOS-RRAM accelerator performs the best in area, energy and speed. Compared to Matlab implementation, it achieves 14.94× speed-up, 447.17× energy-saving and 164.38× area-saving. We also design a 3D-CMOS-ASIC implementation with the similar hardware architecture as 3D multi-layer CMOS-RRAM accelerator with better performance compared to Matlab. The proposed 3D multi-layer CMOS-RRAM 3D accelerator achieves 2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC. To compare the performance with

Table 3.9: Performance comparison under different software and hardware implementations

Implementation	Power, Freq.	Area (mm^2)	Type	Time (s)	Energy (J)
General CPU Processor	130W, 3.46GHz	240, Intel Xeon X5690	Sort	1473.2	191.52
			Mul	736.6	95.76
			IL	729.79	94.87
			L2 norm Div	294.34	38.26
			L2 norm Mul	1667.95	216.83
			OL	750.3	97.54
3D CMOS-ASIC Architecture	1.037W, 1GHz	0.9582, 65nm Global Foundary	Sort	216.65	0.078
			Mul	97.43	3.84
			IL	96.53	3.8
			L2 norm Div	43.29	0.015
			L2 norm Mul	220.62	8.69
			OL	99.25	3.91
			Improvement	7.30×	36.12×
3D CMOS-RRAM Architecture	0.371W, 100MHz	1.026, 65nm CMOS and RRAM	Sort	216.65	0.078
			Mul	22.43	0.293
			IL	22.22	0.291
			L2 norm Div	43.29	0.015
			L2 norm Mul	50.79	0.67
			OL	22.85	0.3
			Improvement	14.94×	447.17×

†6 bit-width configuration is implemented for both CMOS and RRAM. $IL^{\ddagger}$ is for input layer and OL^{*} is for output layer.

GPU, we also implement the same code using Matlab GPU parallel toolbox. It takes 1163.42s for training benchmark CIFAR-10, which is 4.858× faster than CPU. Comparing to our proposed 3D multi-layer CMOS-RRAM architecture, our work achieves 3.07× speed-up and 162.86× energy saving.

3.5 Conclusion

This chapter presents a least-squares-solver based learning method on the single hidden layer neural network. A square-root free Cholesky decomposition technique is applied to reduce the training complexity. Furthermore, the optimized learning algorithm is mapped on CMOS and RRAM based hardwares. The two implementations can be summarized as follows.

- An incremental and square-root-free Cholesky factorization algorithm is introduced with FPGA realization for neural network training acceleration when analyzing the real-time sensed data. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable forecasting accuracy with an average speed-up of 4.56× and 89.05×, when compared to x86 CPU and ARM CPU for inference. Moreover , 450.2×, 261.9× and 98.92× energy saving can be achieved comparing to x86 CPU, ARM CPU and GPU respectively.

- A 3D multi-layer CMOS-RRAM accelerator architecture for machine learning is presented. By utilizing an incremental least-squares-solver, the whole training process can be mapped to the 3D multi-layer CMOS-RRAM accelerator with significant speed-up and energy-efficiency improvement. Experiment results using the benchmark CIFAR-10 show that the proposed accelerator has 2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC hardware implementation; and 14.94× speed-up, 447.17× energy-saving and 164.38× area-saving compared to CPU software implementation. Compared to GPU implementation, our work shows 3.07× speed-up and 162.86× energy-saving.

Chapter 4

Tensor-Solver for Deep Neural Network

4.1 Introduction

The trend of utilizing deeper neural network for machine learning has introduced a grand challenge of high-throughput yet energy-efficient hardware accelerators [6, 7]. For example, the deep neural network in [1] has billions of parameters that requires high computational complexity with large memory usage. Considering the hardware performance, the reduction of memory access is greatly preferred to improve energy efficiency. There is a large power consumption dominated by memory access for data driven applications. For example, under 45-nm CMOS technology, a 32-bit floating point addition consumes only 0.9-pJ whereas a 32-bit SRAM cache-access takes 5.56$\times$ more energy consumption; and a 32-bit DRAM memory access takes 711.11$\times$ more energy consumption [8]. Therefore, simplified neural network by compression is greatly needed for energy-efficient hardware applications.

From the computing algorithm perspective, there are many neural network compression algorithms proposed recently such as connection pruning, weight sharing and quantization [9, 10]. The work in [11] further used low-rank approximation directly to the weight matrix after training. However, the direct approximation of the neural network weight can simply reduce complexity but cannot maintain the accuracy, especially when simplification is performed to the neural network obtained after the training. In contrast, many recent works [12, 105, 106] have found that the accuracy can be maintained when certain constraints (binarization, sparsity, etc) are applied during the training.

From the supporting hardware perspective, the recent in-memory resistive random access memory (RRAM) devices [94, 107, 108, 109] have shown great potential for

an energy-efficient acceleration of multiplication on crossbar. It can be exploited as both storage and computational elements with minimized leakage power due to its non-volatility. Recent researches in [13, 14] show that the 3D heterogeneous integration can further support more parallelism with high I/O bandwidth in acceleration by stacking RRAM on CMOS using through-silicon-vias (TSVs). Therefore, in this chapter, we investigate tensorized neural network from two perspectives, which are summarized as:

- A layer-wise tenorized compression algorithm of multi-layer neural network is proposed. By reshaping neural network weight matrices into high dimensional tensors with a low-rank tensor approximation during training, significant neural network compression can be achieved with maintained accuracy. A corresponding layer-wise training algorithm is further developed for multilayer neural network by modified alternating least-squares (MALS) method. The tensorized neural network (TNN) can provide state-of-the-art results on various benchmarks with significant compression during numerical experiments. For MNIST benchmark, TNN shows 64$\times$ compression rate without accuracy drop. For CIFAR-10 benchmark, TNN shows that compression of 21.57$\times$ compression rate for fully-connected layers with 2.2% accuracy drop.

- A 3D multi-layer CMOS-RRAM accelerator for highly-parallel yet energy-efficient machine learning is proposed in this chapter. A tensor-train based tensorization is developed to represent the dense weight matrix with significant compression. The neural network processing is mapped to a 3D architecture with high-bandwidth TSVs, where the first RRAM layer is to buffer input data; the second RRAM layer is to perform intensive matrix-vector multiplication using digitized RRAM; and the third CMOS layer is to coordinate the reaming control and computation. Simulation results using the benchmark MNIST show that the proposed accelerator has 1.283$\times$ speed-up, 4.276$\times$ energy-saving and 9.339$\times$ area-saving compared to 3D CMOS-ASIC implementation; and 6.37$\times$ speed-up and 2612$\times$ energy-saving compared to 2D CPU implementation.

The rest of this chapter is organized as follows. The shallow and deep tensorized neural networks are discussed in Chapter 4.2 with detailed layer-wise training method. The learning algorithm for TNN and the network interpretation are also elaborated. The detailed implementation on 3D CMOS-RRAM architecture is discussed in Chapter 4.3. Finally, Chapter 4.4 shows the detailed TNN performance on various benchmarks and application results on object recognition and human-action recognition with conclusion drawn in Chapter 4.5.

4.2 Algorithm Optimization

4.2.1 Preliminary

Tensors are natural multi-dimensional generation of matrices. Here, we refer to one-dimensional data as *vectors*, denoted as lowercase bold face letters $\boldsymbol{v}$. Two dimensional arrays are *matrices*, denoted as uppercase bold face letters $\boldsymbol{V}$ and higher dimensional arrays are *tensors* denoted as uppercase bold face calligraphic letters $\boldsymbol{\mathcal{V}}$. To refer to one specific element from a tensor, we use $\boldsymbol{\mathcal{V}}(\boldsymbol{i}) = \boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d)$, where d is the dimensionality of the tensor $\boldsymbol{\mathcal{V}}$ and $\boldsymbol{i}$ is the index vector. The relationship between tensors and matrices are shown in Fig. 4.1. A summary of notations and descriptions is shown in Table 4.1 for clarification.

For a large high dimensional tensor, it is not explicitly represented but represented as some low-parametric format. A canoical decomposition is one such low parametric data format. A canonical decomposition of d-dimensional $n_1 \times n_2 \times ... \times n_d$ tensor $\boldsymbol{\mathcal{V}}$ is a decomposition of $\boldsymbol{\mathcal{D}}$ as a linear combination of a minimal number of rank-1 terms

$$\boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d) = \sum_{\alpha=1}^{r} \boldsymbol{U}_1(i_1, \alpha)\boldsymbol{U}_2(i_2, \alpha)...\boldsymbol{U}_d(i_d, \alpha) \tag{4.1}$$

where r is defined as the tensor $\boldsymbol{\mathcal{D}}$ rank and the matrices $\boldsymbol{U}_k = [\boldsymbol{U}_k(i_k, \alpha)]$ are called canonical factors. This data format is very efficient for data storage. It requires only $O(dnr)$ instead of $O(n^d)$ to store $\boldsymbol{\mathcal{V}}$. However, there are a few drawbacks on this decomposition. The computation of the tensor rank r is proved to be an NP-hard problem [110]. Even the approximation of canoical decomposition with a fixed rank k can not guarantee to work well. Therefore, alternative representations such tucker decomposition and tensor-train decomposition are developed.

Tucker decomposition [110, 111] is a high-order singular value decomposition (SVD) and very stable. However, it is mainly designed for the 3-dimensional matrix and the number of parameters is relatively large, $O(dnr + r^d)$. Therefore, this decomposition method is not considered for model compression.

A d-dimensional $n_1 \times n_2 \times ... \times n_d$ tensor $\boldsymbol{\mathcal{V}}$ is decomposed into the tensor-train data format if tensor core $\boldsymbol{G}_k$ is defined as $r_{k-1} \times n_k \times r_k$ and each element is defined [110] as

$$\begin{aligned} \boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d) = \sum_{\alpha_0, \alpha_1 ... \alpha_d}^{r_0, r_1, ...r_d} & \boldsymbol{G}_1(\alpha_0, i_1, \alpha_1) \\ & \boldsymbol{G}_2(\alpha_1, i_2, \alpha_2)...\boldsymbol{G}_d(\alpha_{d-1}, i_d, \alpha_d) \end{aligned} \tag{4.2}$$

where α_k is the index of summation, which starts from 1 and stops at rank r_k. $r_0 = r_d$

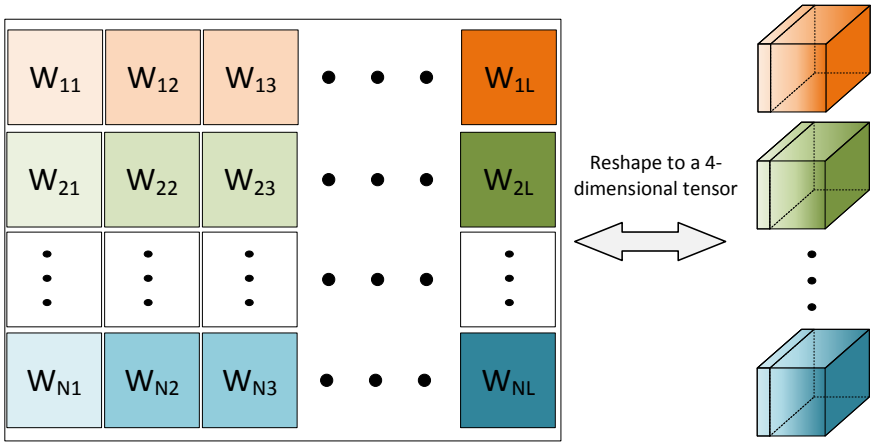


Figure 4.1: Block matrices to a 4-dimensional tensors

Table 4.1: Symbol notations and detailed descriptions.

Notations	Descriptions
$\boldsymbol{\mathcal{V}} \in \mathbb{R}^{n_1 \times n_2 \times ... \times n_d}$	d-dimensional tensor of size $n_1 \times n_2 ... n_d$
$\boldsymbol{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$	Tensor cores of tensor-train data format
$\boldsymbol{W}_1, \boldsymbol{W}_2, ..., \boldsymbol{W}_{d-1}$	Neural network weights of d layers
$\boldsymbol{B}_1, \boldsymbol{B}_2, ..., \boldsymbol{B}_{d-1}$	Neural network bias of d layers
$\boldsymbol{H}_1, \boldsymbol{H}_2, ..., \boldsymbol{H}_{d-1}$	Activation matrix of d layers
$\boldsymbol{T}, \boldsymbol{Y}$	Labels and neural network output
$\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{V}$	SVD decomposition matrices
$\boldsymbol{X}$	Input features
$r_0, r_1, ..., r_d$	Rank of tensor cores
$i_1, i_2, ..., n_d$	Index vectors referring to tensor element
$n_1, n_2, ..., n_d$	Mode size of tensor $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{n_1 \times n_2 \times ... \times n_d}$
$p(1/y), ..., p(m/y)$	Predicted probability on each class
n_m	Maximum mode size of $n_1, n_2, ..n_d$
N_t	Number of training samples
N	Number of input features
M	Number of classes

= 1 is for the boundary condition and $n_1, n_2, ...n_d$ are known as mode size. Here, r_k is the core rank and $\boldsymbol{G}$ is the core for this tensor decomposition. By using the notation of $\boldsymbol{G}_k(i_k) \in \mathbb{R}^{r_{k-1} \times r_k}$, we can rewrite the above equation in a more compact way.

$$\boldsymbol{\mathcal{V}}(i_1, i_2, ...i_d) = \boldsymbol{G}_1(i_1)\boldsymbol{G}_2(i_2)...\boldsymbol{G}_d(i_d) \tag{4.3}$$

where $\boldsymbol{G}_k(i_k) \in \mathbb{R}^{r_{k-1} \times r_k}$ is a slice from the 3-dimensional matrix $\boldsymbol{G}_k$. Fig. 4.2 shows the general idea of tensorized neural network. A two-dimensional weight is folded into a three-dimensional tensor and then decomposes into tensor cores $\boldsymbol{G}_1, \boldsymbol{G}_2, ...\boldsymbol{G}_d$. These

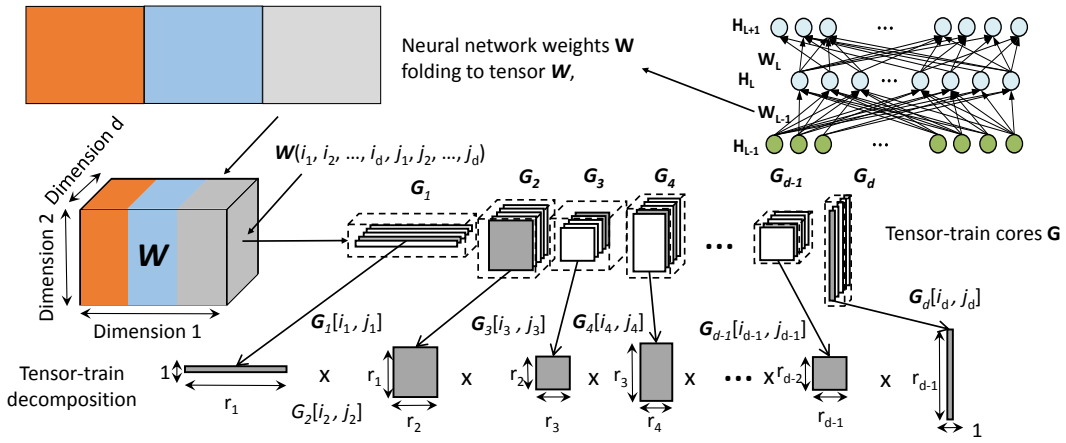


Figure 4.2: Neural network weight tensorization and represented by the tensor-train data format for parameter compression (from n^d to dnr^2)

tensor cores are relatively small 3-dimensional matrices due to small value of rank r, resulting a high neural network compression rate.

Such representation is memory efficient to store high dimensional data. For example, a d-dimensional tensor requires $n_1 \times n_2 \times ... \times n_d = n^d$ parameters. However, if it is represented using the tensor-train format, it takes only $\sum_{k=1}^{d} n_k r_{k-1} r_k$ parameters. So if we manage to reduce the rank of each core, we can efficiently represent data with high compression rate and store them distributively.

In this section, we will discuss a tensor-train formatted neural network during the training, which can achieve high compression rate yet with maintained accuracy. Based on the tensor-train data format, we develop a shallow tensorized neural network (TNN). Furthermore, a stacked auto-encoder based deep TNN is developed to tackle more complicated tasks and achieve high compression rate.

4.2.2 Shallow Tensorized Neural Network

We first start with a single hidden layer feed forward neural network [112] and later extend to multi-layer neural network [112, 113, 114]. We refer to one hidden layer neural network as shallow neural networks and more hidden layers as deep neural networks. Furthermore, we define a tensorized neural network (TNN) if the weight of the neural network can be represented in the tensor-train data format. For example, a two-dimensional weight $\boldsymbol{W} \in \mathbb{R}^{L \times N}$ can be reshaped to a $k_1 + k_2$ dimensional tensor by factorizing $L = \prod_{d=1}^{k_1} l_d$ and $N = \prod_{d=1}^{k_2} n_d$ and such tensor can be further decomposed into the tensor-train data format.

As shown in Fig. 4.3, we can train a single hidden layer neural network based on

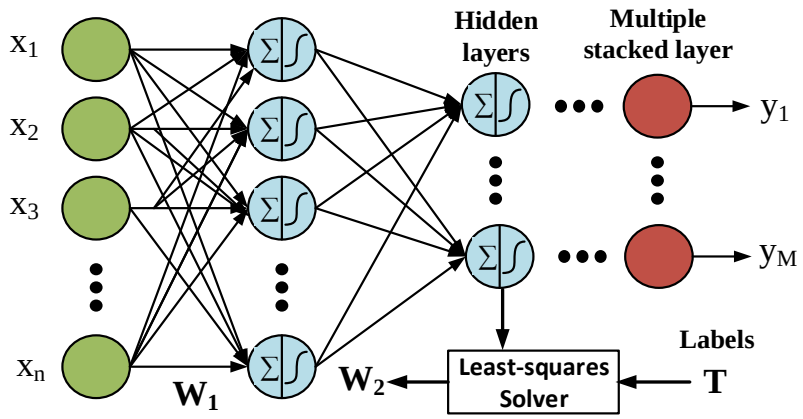


Figure 4.3: Layer-wise training of neural network with least-squares solver

data features $\boldsymbol{X}$ and training labels $\boldsymbol{T}$ with N_t number of training samples, N dimensional input features and M classes. During the training, one needs to minimize the error function with determined weights $\boldsymbol{W}$ and bias $\boldsymbol{B}$:

$$E = ||\boldsymbol{T} - f(\boldsymbol{W}, \boldsymbol{B}, \boldsymbol{X})||_2 \tag{4.4}$$

where $f(\cdot)$ is the trained model to perform the predictions from input.

Here, we discuss one general machine learning algorithm using least-squares learning without tensor-train based weights, which is mainly inspired from [82]. Then we show how to train a tensorized neural network. We firstly build the relationship between hidden neural nodes and input features as

$$\boldsymbol{preH} = \boldsymbol{X}\boldsymbol{W}_1 + \boldsymbol{B}_1, \; \boldsymbol{H}_1 = \frac{1}{1 + e^{-\boldsymbol{preH}}} \tag{4.5}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{N \times L}$ and $\boldsymbol{B}_1 \in \mathbb{R}^{N_t \times L}$ are randomly generated input weight and bias formed by w_{ij} and b_{ij} between $[-1, 1]$. The training process is designed to find $\boldsymbol{W}_2$ such that we can minimize:

$$\arg\min_{\boldsymbol{W}_2} \; ||\boldsymbol{H}_1\boldsymbol{W}_2 - \boldsymbol{T}||_2 + \lambda ||\boldsymbol{W}_2||_2 \tag{4.6}$$

where $\boldsymbol{H}_1$ is the hidden-layer output matrix generated from the Sigmoid function for activation; and λ is a user defined parameter that biases the training error and output weights. The output weight $\boldsymbol{W}_2$ is computed based on least-squares problem [115]:

$$\boldsymbol{W}_2 = (\boldsymbol{H}_1^T\boldsymbol{H}_1 + \lambda \boldsymbol{I})^{-1}\boldsymbol{H}_1^T\boldsymbol{T} \tag{4.7}$$

The benefits of minimizing the norm of weight $\boldsymbol{W}_2$ can be summarized as follows.

Firstly, the additional constraint of minimizing the norm guarantees that we can find a solution for the optimization problem. If $\boldsymbol{H}_1^T\boldsymbol{H}_1$ is close to singular, the new added $\lambda\boldsymbol{I}$ guarantee that $\boldsymbol{H}_1^T\boldsymbol{H}_1+\lambda\boldsymbol{I}$ is invertible. Secondly, it prevents overfitting and improves generality. Minimizing the norm of weight matrix $\boldsymbol{W}_2$ will help generate a more stable model. For example, if the value of some coefficients are very large, a small noise may significant change the predicted result, which is not desirable. We can also convert it into a standard least-squares solution:

$$\begin{aligned} &\boldsymbol{W}_2 = (\tilde{\boldsymbol{H}}_1^T\tilde{\boldsymbol{H}}_1)^{-1}\tilde{\boldsymbol{H}}_1{}^T\tilde{\boldsymbol{T}},\ \tilde{\boldsymbol{H}}_1 \in \mathbb{R}^{N_t\times L} \\ &\textit{where } \tilde{\boldsymbol{H}}_1 = \begin{pmatrix} \boldsymbol{H}_1 \\ \sqrt{\lambda}\boldsymbol{I} \end{pmatrix}\ \ \tilde{\boldsymbol{T}} = \begin{pmatrix} \boldsymbol{T} \\ \boldsymbol{0} \end{pmatrix} \end{aligned} \tag{4.8}$$

where $\tilde{\boldsymbol{T}} \in \mathbb{R}^{(N_t+L)\times M}$ and M is the number of classes. $\boldsymbol{I} \in \mathbb{R}^{L\times L}$ and $\tilde{\boldsymbol{H}}_1 \in \mathbb{R}^{(N_t+L)\times L}$. Then neural network output will be

$$\begin{aligned} \boldsymbol{Y} &= f(\boldsymbol{W},\boldsymbol{B},\boldsymbol{X}_t) \\ p(i/y_i) &\approx y_i,\ y_i \in \boldsymbol{Y} \end{aligned} \tag{4.9}$$

where $\boldsymbol{X}_t$ is the inference data and i represents class index $i \in [1,\ M]$. We approximate the prediction probability for each class by the output of neural network.

For TNN, we need to tensorize input weight $\boldsymbol{W}_1$ and output weight $\boldsymbol{W}_2$. Since the input weight $\boldsymbol{W}_1$ is randomly generated, we can also randomly generate tensor core $\boldsymbol{G}$, then create tensorized input weight $\boldsymbol{\mathcal{W}}_1$ based on (4.3). For output weight $\boldsymbol{W}_2$, it requires to solve a least-squares problem in the tensor-train data format. This is solved using the modified alternating least squares method and will be discussed in Chapter 4.2.4.

After tensorization, the neural network processing is a direct application of tensor-train-matrix-by-vector operations. Tensorized weights $\boldsymbol{\mathcal{W}}_1 \in \mathbb{R}^{n_1l_1\times n_2l_2\times..\times n_dl_d}$ is equivalent to $\boldsymbol{W}_1 \in \mathbb{R}^{N\times L}$, where n_k and l_k following $N=\prod_{k=1}^d n_k$ and $L=\prod_{k=1}^d l_k$. The neural network forward pass computation in the tensor-train data format is

$$\boldsymbol{\mathcal{H}}_1(\boldsymbol{i}) = \sum_{j_1,j_2,...,j_d}^{l_1,l_2,...,l_d} \boldsymbol{\mathcal{X}}(\boldsymbol{j})\boldsymbol{G}_1[i_1,j_1]\boldsymbol{G}_2[i_2,j_2]...\boldsymbol{G}_d[i_d,j_d] + \boldsymbol{\mathcal{B}}_1(\boldsymbol{i}) \tag{4.10}$$

where $\boldsymbol{i} = i_1,i_2,...,i_d, i_k \in [1,n_k]$, $\boldsymbol{j} = j_1,j_2,...,j_d,j_k \in [1,l_k]$ and $\boldsymbol{G}[i_d,j_d] \in \mathbb{R}^{r_{k-1}\times r_k}$ is a slice of tensor cores. This is d times summation with summation index j_k increased from 1 to l_k. We use a pair $[i_k,j_k]$ to refer to the index of the mode n_kl_k. This tensor-train-matrix-by-vector multiplication complexity is $O(dr^2n_m\ max(N,L))$, where r is the maximum rank of cores $\boldsymbol{G_i}$ and n_m is the maximum mode size of tensor $\boldsymbol{\mathcal{W}}_1$. This

Algorithm 3 Single hidden layer training of TNN using modified alternating least-squares method

Input: Input Set ($\boldsymbol{X} \in \mathbb{R}^{N_t \times N}, \boldsymbol{T} \in \mathbb{R}^{N_t \times M}$), $\boldsymbol{X}$ is the input data and $\boldsymbol{T}$ is the desired output (labels, etc) depending on the layer architecture, activation function $G(a_i, b_i, x_j)$, Number of hidden neural nodes L and accepted training accuracy Acc.

Output: Neural Network output weight $\boldsymbol{\mathcal{W}}_2$

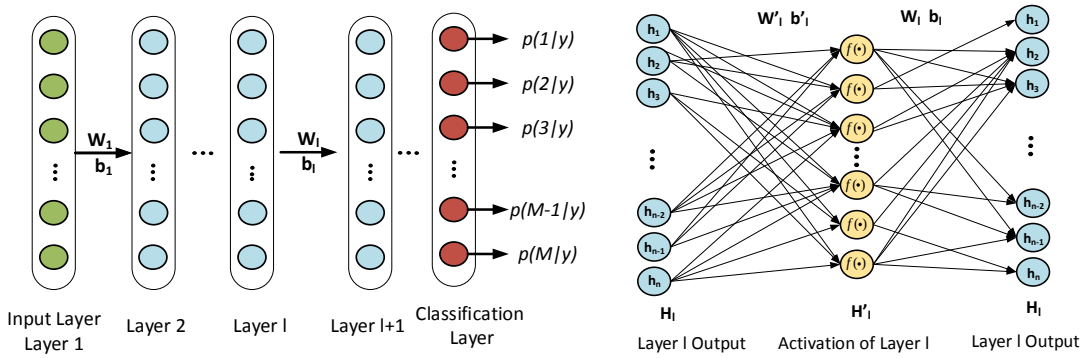
1: Tensorization: Factorize $N = \prod_{k=1}^{d} n_k$ and $L = \prod_{k=1}^{d} l_k$ where N and L are dimensions of the input weight $\boldsymbol{W}_1 \in \mathbb{R}^{N \times L}$. The tensorized input weight $\boldsymbol{\mathcal{W}}_1 \in \mathbb{R}^{n_1 l_1, n_2 l_2, ..., n_d l_d}$.
2: As (4.2) indicates, we need generate d cores $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, ...$\boldsymbol{G}_d$, where each core follows $\boldsymbol{G}_i \in \mathbb{R}^{r_{i-1} \times n_i l_i \times r_i}$. Please note that since we prefer random input weights, $\boldsymbol{G}_i$ is randomly generated with small rank r_{i-1} and r_i.
3: Perform Tensor-train-matrix-by-matrix $\boldsymbol{preH} = \boldsymbol{X}\boldsymbol{W}_1 + \boldsymbol{B}_1$
4: Activation function $\boldsymbol{H}_1 = \mathbf{1}/(\mathbf{1} + e^{-\boldsymbol{preH}})$.
5: Calculate the output weight $\boldsymbol{\mathcal{W}}_2$ by modified-alternating-least-squares (MALS), which is equivalent to for matrix calculation $\boldsymbol{W_2} = (\boldsymbol{H}_1^T \times \boldsymbol{H}_1)^{-1} \times \boldsymbol{H}_1^T \boldsymbol{T}$
6: Calculate the training accuracy Acc. If it is less than required, increase L and repeat from step 1.
7: END Training

can be illustrated in Fig. 4.2, where a two-dimensional weight is folded into a three-dimensional tensor and then decomposed into tensor cores $\boldsymbol{G}_1, \boldsymbol{G}_2, ...\boldsymbol{G}_d$. The pair $[i_k, j_k]$ refers to a slice of the tensor core $\boldsymbol{G}_k$. This can be very efficient if the rank r is very small compared to general matrix-vector multiplication. It is also favorable for distributed computation since each core is small and matrix multiplication is associative.

Algorithm 3 summarizes the whole training process of a single hidden layer tensorized neural network, which also provides the extension to deep neural network. Step 1 determines the mode size of the d-dimensional tensor $\boldsymbol{\mathcal{W}}_1$. Then based on the mode size of each dimension, weight tensor cores are randomly generated. Tensor-train-matrix-by-matrix multiplication and modified-alternating-least-squares method are performed to find the output weight $\boldsymbol{\mathcal{W}}_2$. Please note that tensor $\boldsymbol{\mathcal{W}}_2$ can be further compressed by left-to-right sweep using truncated singular value decomposition (SVD) method. More details will be discussed in Chapter 4.2.3.

4.2.3 Deep Tensorized Neural Network

In the conventional training method, the weight in the multi-layer neural network is randomly initialized. The layer-by-layer training is to initialize the weights of the multi-layer deep neural network using the auto-encoder method. To build a multi-layer neural network as shown in Fig. 4.4(a), we propose a layer-wise training process based on stack auto-encoder. An auto-encoder layer is to set the single layer output $\mathbf{T}$ the same as



(a) (b)

Figure 4.4: (a) Deep Neural Network (b) Layer-wise training process for deep neural network

input $\mathbf{X}$ and find an optimal weight to represent itself [113, 116, 117]. By stacking auto-encoder layers on the finial decision layer, we can build the multi-layer neural network. Therefore, Algorithm 3 can also be viewed as unsupervised layer-wise learning based on auto-encoder by changing the desired output $\boldsymbol{T}$ into input features $\boldsymbol{X}$.

Our proposed TNN applies layer-by-layer learning as shown in Fig. 4.4(b). The learning process is the minimization of

$$\arg\min_{\boldsymbol{W}_l} = ||f(f(\boldsymbol{H}_l\boldsymbol{W}'_l+\boldsymbol{B}'_l)\boldsymbol{W}_l+\boldsymbol{B}_l)-\boldsymbol{H}_l||_2 \tag{4.11}$$

where $\boldsymbol{W}_l$ is the auto-decoder learned weights and will be passed to the multi-layer neural network on layer L. $\boldsymbol{W}'_l$ and $\boldsymbol{B}'_l$ are the random generated input weights and bias respectively for such auto-encoder. $f(\cdot)$ is the activation function. The rest layers are initialized in the same fashion by the auto-encoder based learning process. Algorithm 4 summarizes the whole training process of multi-layer tensorized neural network. The auto-encoder is to reconstruct the noisy input with certain constraints and the desired output is set to input itself as shown in Fig. 4.4(b). Therefore, the first training process is to perform auto-encoder layer-by-layer. Then the last layer is performed by modified-alternating-least-squares (MALS) method as shown in Algorithm 3. The optimization process of MALS will be discussed later in Chapter 4.2.4

Convolutional neural network (CNN) is widely applied for image recognitions. The process of convolving a signal can be mathematically defined as [1]

$$\mathcal{Y}_{i''j''d''} = \sum_{i'=1}^{H'}\sum_{j'=1}^{W'}\sum_{d'=1}^{D}\mathcal{F}_{i'j'd'd''} \times \mathcal{X}_{i''+i'-1,\ j''+j'-1,\ d'} \tag{4.12}$$

[1] Stride is assumed 1 and there is no padding on the input data

Algorithm 4 Stacked auto-encoder based layer-wise training of multi-layer TNN

Input: Input Set $(\boldsymbol{X}, \boldsymbol{T})$, $\boldsymbol{X}$ is the input data and $\boldsymbol{T}$ is label matrix, NL number of layers , activation function $G(a_i, b_i, x_j)$, maximum number of hidden neural nodes L and accepted training accuracy Acc.

Output: Neural Network output weight β

While $l < NL - 1$
Prepare auto-encoder training set $\boldsymbol{H}_{nl}$, L, Acc, where $\boldsymbol{H}_l = \boldsymbol{X}$ for the first layer.
perform least-squares optimization on layer l $\arg\min_{W_l} = ||f(f(\boldsymbol{H}_l\boldsymbol{W}'_l + \boldsymbol{B}'_l)\boldsymbol{W}_l + \boldsymbol{B}_l) - \boldsymbol{H}_l||_2$
Calculate next layer output $\boldsymbol{H}_{l+1} = f(\boldsymbol{H}_l, \boldsymbol{W}_l, \boldsymbol{B}_l)$
END While
For the final layer, prepare the feed forward intermediate results $\boldsymbol{H}_{NL-1}$, and label $\boldsymbol{T}$.
Perform Algorithm 3 training process for the final layer weight $\boldsymbol{W}_{NL-1}$
If the training accuracy is less than required Acc, perform BP based on (4.13)
END Training

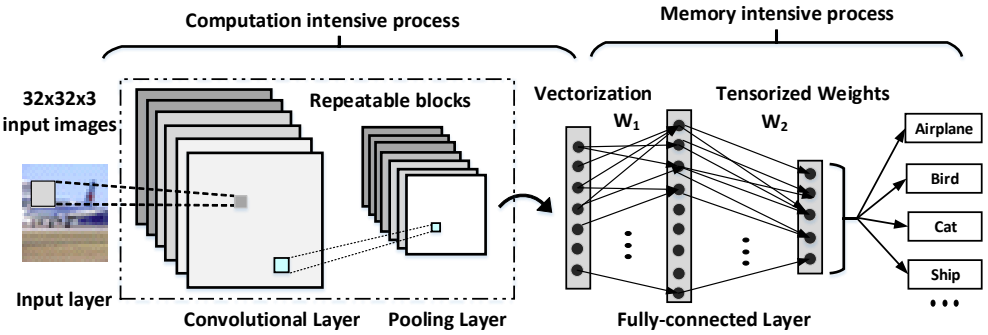


Figure 4.5: Convolution neural network with tensorized weights on fully-connected layers for neural network compression

where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{H \times W \times D}$ is the input, $\boldsymbol{\mathcal{F}} \in \mathbb{R}^{H' \times W' \times D \times D"}$ is the filter and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{H'' \times W'' \times D''}$ is the output from this convolution layer. H, W and D are input data height, width and channels respectively whereas H', W' and D' represent kernels specification accordingly. A bias term $\boldsymbol{B}$ can be added to (4.12) but we omit this for clearer presentation. A fully-connected layer is a special case of convolution operation and it is defined when the output map has dimensions $W'' = H'' = 1$. For tensorized CNN, we will treat CNN as feature extractor and use TNN to replace the fully-connected layer to compress the neural network since the parameters from fully-connected layers consume significant large potions of parameters. For example, a VGG-16 net [54] has 89.36% $(123.64M/148.36M)$ parameters using in the 3 fully-connected layers. As shown in Fig. 4.5, repeated blocks of convolutional layer, pooling layer and activation layer are added on the top. We regard this as CNN feature extractor. Then these features are used in TNN as input features with labels for further training. Finally, the whole network can be fine-tuned with backward propagation (BP).

In this section, we will firstly discuss the layer-wise training process of TNN using

the modified alternating least squares method. Then, a backward propagation based fine-tuning is elaborated. To achieve a higher compression rate, a non-uniform quantization on tensor cores is proposed. Finally, the TNN network interpretation is discussed, which greatly fits in the current deep learning framework.

4.2.4 Layer-wise Training of TNN

Layer-wise training of TNN requires to solve a least-squares problem in the tensor-train data format. For the output weight $\boldsymbol{W}_2$ in (4.6), direct tensorization of $\boldsymbol{W}_2$ is similar to high-order SVD and performance degradation is significant with relative small compression rate. Instead, we propose a tensor-train based least-squares training method using modified alternating least squares algorithm (also known as density matrix renormalization group in quantum dynamics) [74, 118]. The modified alternating least squares is to find optimal tensor cores by performing least-squares optimization of one tensor cores and fixing the rest tensor cores. For example, when we optimize $\boldsymbol{G}_1$, we will fix $\boldsymbol{G}_2$, $\boldsymbol{G}_3$, ..., $\boldsymbol{G}_N$ and perform the least squares optimization. The modified alternating least squares (MALS) for minimization of $||\boldsymbol{H}_1\boldsymbol{\mathcal{W}}_2 - \boldsymbol{T}||_2$ is working as follows.

1. **Initialization**: Randomly initialized cores $\boldsymbol{G}$ and set $\boldsymbol{\mathcal{W}}_2 = \boldsymbol{G}_1 \times \boldsymbol{G}_2 \times ... \times \boldsymbol{G}_d$
2. **Sweep of Cores**: core $\boldsymbol{G}_k$ is optimized with other cores fixed. Left-to-right sweep from $k = 1$ to $k = d$
3. **Supercore generated**: Create supercore $\boldsymbol{X}(k,k+1) = \boldsymbol{G}_k \times \boldsymbol{G}_{k+1}$ and find it by minimizing least-squares problem $||\boldsymbol{H}_1 \times \boldsymbol{Q}_{k-1} \times \boldsymbol{X}_{k,k+1} \times \boldsymbol{R}_{k+2} - \boldsymbol{T}||_2$, reshape $\boldsymbol{Q}_{k-1} = \prod_{i=1}^{k-1} \boldsymbol{G}_i$ and $\boldsymbol{R}_{k+2} = \prod_{i=k+2}^{d} \boldsymbol{G}_i$ to fit matrix-matrix multiplication
4. **Split supercore**: SVD $\boldsymbol{X}(k,k+1) = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$, let $\boldsymbol{G}_k = \boldsymbol{U}$ and $\boldsymbol{G}_{k+1} = \boldsymbol{S}\boldsymbol{V}^T \times \boldsymbol{G}_{k+1}$. $\boldsymbol{G}_k$ is determined and $\boldsymbol{G}_{k+1}$ is updated. Truncated SVD can also be performed by removing smaller singular values to reduce ranks.
5. **Sweep Termination**: Terminate if maximum sweep times reached or error is smaller than required.

The low rank initialization is very important to have smaller rank r for each core. Each supercore generation is the process of solving least-squares problems. The complexity of least-squares for $\boldsymbol{X}$ are $O(n_m R r^3 + n_m^2 R^2 r^2)$ [118] and the SVD compression requires $O(n_m r^3)$, where R, r and n_m are the rank of activation matrix $\boldsymbol{H}_1$, the maximum rank of core $\boldsymbol{G}$ and maximum mode size of $\boldsymbol{\mathcal{W}}_2$ respectively. By using truncated SVD, we can adaptively reduce the rank of each core to reduce the computation complexity.

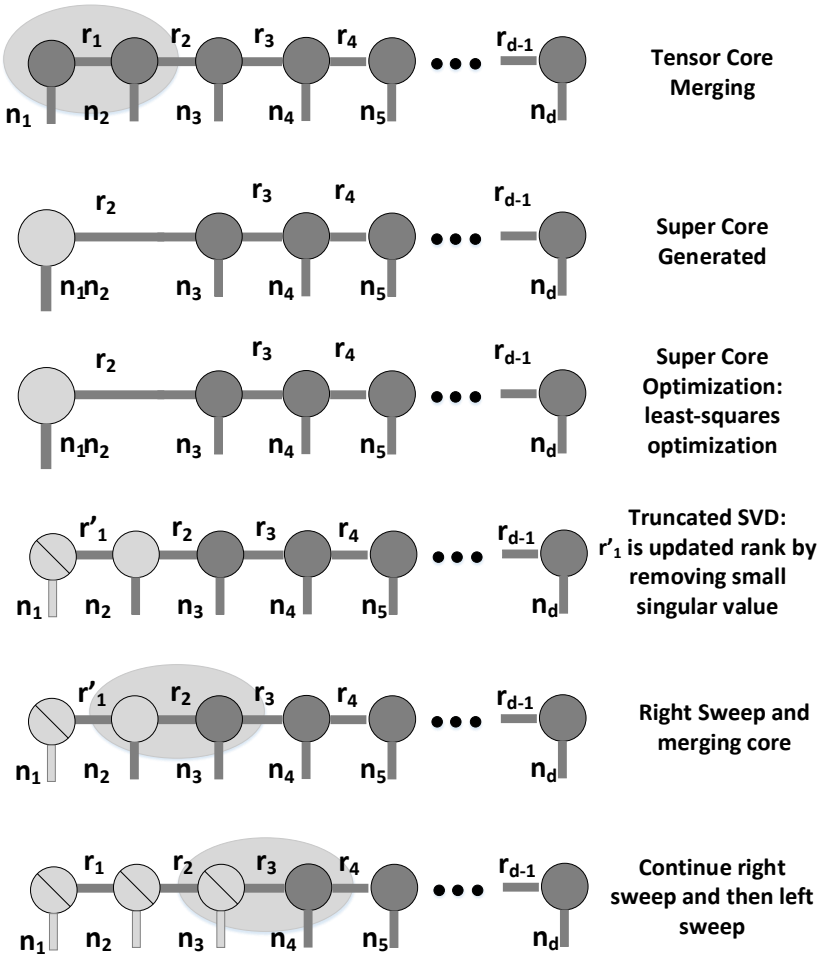


Figure 4.6: Diagrammatic notation of tensor-train and the optimization process of modified alternating least-squares method

Fig. 4.6 gives an example of MALS sweeping algorithms on diagrammatic notation of tensors [2]. Firstly, we perform a random guess of tensor cores $\boldsymbol{G}$, which is represented by a node with edges. The edge represents the dimension of tensor cores. So the most left one and right one have two edges and the rest have thee edges. This is exactly the same as tensor core definition $\boldsymbol{G} \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ with boundary condition $r_0 = r_d = 1$ as discussed in Chapter 4.2.1. Then, adjacent cores merges together to form a suprecore. Such supercore is optimized by standard least-squares optimization. After that, the supercore is spited into two cores by truncated SVD, where small singular values are removed. Finally, the sweep will continue from left to right and then sweep back until the algorithm meets the maximum sweeping times or satisfies the minimum error requirement.

[2]Diagrammatic notation of tensors is detailed in [74].

4.2.5 Fine-tuning of TNN

End-to-end learning of TNN is desirable to further improve the accuracy of the layer-wise learned network. Backward propagation (BP) is widely used to train deep neural network. For TNN, the gradient of tensor layer can be computed as

$$\frac{\partial E}{\partial \boldsymbol{G}_k} = \sum_{\boldsymbol{i}} \frac{\partial E}{\partial \boldsymbol{\mathcal{H}}(\boldsymbol{i})} \frac{\partial \boldsymbol{\mathcal{H}}(\boldsymbol{i})}{\partial \boldsymbol{G}_k} \tag{4.13}$$

where E is the loss function and $\boldsymbol{\mathcal{H}}(\boldsymbol{i})$ has the same definition as (4.10). The computation complexity is very high ($O(d^4 r^2 m\ max(M,N))$), which increases the training time and limits the number of epochs. Therefore, it is necessary to have a good initialization by our proposed layer-wise learning method to reduce the number of epochs required for BP.

4.2.6 Quantization of TNN

To achieve high compression rate without accuracy loss, we further show how to use less bit-width to represent weights in the tensor-train data format. Instead of performing quantization on weight $\boldsymbol{W}$, we propose a non-uniform quantization on tensor cores $\boldsymbol{G}$. Let X represent the vectorized tensor-train cores $\boldsymbol{G}$ and $\hat{X}$ be the representative levels. Given a probability density function (pdf) $f_X(x)$, our objective is to find

$$\min_{\hat{X}} MSE = E[(X - \hat{X})^2] \tag{4.14}$$

where $\hat{X}$ are the quantized representative levels. This can be solved by iterative optimization for quantizer design. Firstly, we can have a random guess of representative levels $\hat{x}_q$. Then we can calculate the decision thresholds as $t_q = (\hat{x}_q + \hat{x}_{q-1})/2$, where $q = 1, 2, 3, ..., M_q - 1$. M_q represents the number of levels. The new representative values can be calculated as

$$\hat{x}_q = \frac{\int_{t_q}^{t_{q+1}} x f_X(x) dx}{\int_{t_q}^{t_{q+1}} f_X(x) dx} \tag{4.15}$$

We can iteratively calculate the decision thresholds and also the new representative values until convergence is reached for the optimal quantizer. Note that we can estimate the pdf $f_X(x)$ by sampling tensor core weight values. Detailed results will be shown in the experiments.

4.2.7 Network Interpretation of TNN

The tensor-train based neural network can be greatly fit into the multilayer neural network framework. We will explain this from both deep features perspective and stacked auto-encoders perspective. By representing weights in tensor-train data format, we actually approximate deep neural network architecture in a compressed method. Obviously, the tensor cores are not unique and can be orthogonalized by left-to-right sweep or right-to-left sweep. This sweep is achieved by performing singular vector decomposition (SVD) to tensor cores:

$$\boldsymbol{G} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T \tag{4.16}$$

where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal and $\boldsymbol{S}$ are the singular values matrix. So for left-to-right sweep, we can keep $\boldsymbol{U}$ and merge $\boldsymbol{S}\boldsymbol{V}^T$ to the next core. This process can be explained as

$$\begin{aligned}\boldsymbol{\mathcal{W}}(i_1, i_2, ...i_d) =& \boldsymbol{G}_1(i_1)\boldsymbol{G}_2(i_2)...\boldsymbol{G}_d(i_d) \\ & \boldsymbol{U}_1\boldsymbol{S}_1\boldsymbol{V}_1^T\boldsymbol{G}_2(i_2)...\boldsymbol{G}_d(i_d) \\ & \boldsymbol{U}_1\boldsymbol{U}_1\boldsymbol{U}_2...\boldsymbol{U}_{d-1}\boldsymbol{C}\end{aligned} \tag{4.17}$$

where $\boldsymbol{U}_1, \boldsymbol{U}_2, ...$ are orthogonal cores by SVD operations and $\boldsymbol{C}$ is the final core for this weights. For such neural network weight $\boldsymbol{\mathcal{W}}$, it means the input features have performed orthogonal transformations using $\boldsymbol{U}_1, \boldsymbol{U}_2...$ and then by multiplying $\boldsymbol{C}$, the feature space will be projected to a large or smaller space. If we view each orthogonal transformation $\boldsymbol{U}$ as one layer of neural network without activation function, the tensor-train based weights indeed represent many weights of a deep nerual network architecture.The MALS process are equivalently to backward propagation except it is one-step weight updates [3] and each sweep works as epoch in backward propagation learning algorithms.

From the stacked auto-encoder perspective, the optimization problem for the L-th layer is

$$\arg\min_{\boldsymbol{W}_L} = ||f(\boldsymbol{H}_L\boldsymbol{W}'_L)\boldsymbol{W}_L - \boldsymbol{H}_L||_2 \tag{4.18}$$

where $\boldsymbol{W}'_L$ is randomly generated and $f(\cdot)$ is the activation function. $\boldsymbol{W}_L$ is the auto-encoder computed weight to initialize the L-th layer weight of the multi-layer neural network. If the activation function is removed, the final training objective after auto-encoder can be represented as

$$\arg\min_{\boldsymbol{W}_1,...,\boldsymbol{W}_L} = ||\boldsymbol{W}_1\boldsymbol{W}_2, ..., \boldsymbol{W}_L\boldsymbol{W}_f\boldsymbol{X} - \boldsymbol{T}||_2 \tag{4.19a}$$

$$\arg\min_{\boldsymbol{\mathcal{W}}} = ||\boldsymbol{\mathcal{W}}\boldsymbol{W}_f\boldsymbol{X} - \boldsymbol{T}||_2 \tag{4.19b}$$

[3] Provided the loss function is euclidean distance.

where $\boldsymbol{W}_f$ represents final decision layer, which is not determined from auto-encoder. (4.19a) is equivalent to find a tensor $\boldsymbol{\mathcal{W}}$ with tensor-train decomposition cores $\boldsymbol{W}_1, \boldsymbol{W}_2 ... \boldsymbol{W}_L$ as shown in (4.19b). Under such interpretation, we expect similar behavior of tensorized neural network as stacked auto-encoder based neural networks.

4.3 Hardware Implementation

In this section, the 3D hardware platform is proposed based on the non-volatile RRAM-crossbar devices with the design flow for TNN mapping on the proposed architecture. The RRAM-crossbar device is already introduced in Chapter 3.3.2.

4.3.1 3D Multi-layer CMOS-RRAM Architecture

3D-Integration: Recent works [95, 119] show that the 3D integration supports heterogeneous stacking because different types of components can be fabricated separately with different technologies and then layers can be stacked into 3D structure. Therefore, stacking non-volatile memories on top of microprocessors enables cost-effective heterogeneous integration. Furthermore, works in [96, 97, 120] also show the feasibility to stack RRAM on CMOS to achieve smaller area and lower energy consumption.

3D-stacked Modeling: In this proposed accelerator, we adopt the face-to-back bonding with TSV connections. TSVs can be placed vertically on the whole layer as shown in Fig. 4.7. The granularity at which TSV can be placed is modeled based on CACTI-3DD using the fine-grained strategy [121], which will automatically partition the memory array to utilize TSV bandwidth. Although this strategy requires a large number of TSV, it provides higher bandwidth, better access latency and smaller power, which is greatly needed to perform highly-parallel tensor based computation. We use this model to evaluate our proposed architecture and will show the bandwidth improvement in Chapter 4.4.3.

Architecture: In this section, we propose a 3D multi-layer CMOS-RRAM accelerator with three layers as shown in Fig. 4.7. This accelerator is composed of a two-layer RRAM-crossbar and a one-layer CMOS circuit. More specifically, they are designed as follows.

- Layer 1 of RRAM-crossbar is implemented as a buffer to temporarily store input data and neural network model weights as shown in Fig. 4.7(a). The tensor cores are 3-dimensional matrices and each slice is a 2-dimensional matrix stored distributively in a H-tree like fashion on the layer 1 as described in Fig. 4.7(b).

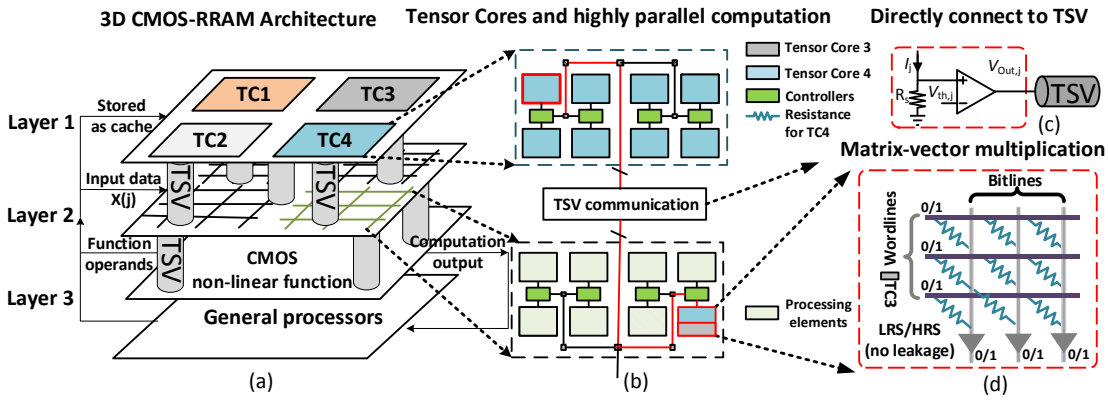


Figure 4.7: (a) Proposed 3D multi-layer CMOS-RRAM accelerator (b)RRAM memory and highly parallel RRAM based computation engine (c) TSV communication (d) Memristor Crossbar

They can be accessed through TSV as the input of the RRAM-Crossbar or used to configure the RRAM crossbar resistance for Layer 2.

- Layer 2 of RRAM-crossbar performs logic operations such as matrix-vector multiplication and also vector addition. As shown in Fig. 4.7(b), layer 2 collect tensor cores from layer 1 through TSV communication to perform parallel matrix-vector multiplication. The RRAM data is directly sent through TSV. The word line takes the input (in this case, tensor core 3) and the multiplicand (in this case, tensor core 4) is stored as the conductance of the RRAM. The output will be collected from the bit lines as shown in Fig. 4.7(b).

- Layer 3 is designed to perform the overall synchronization of the tensorized neural network. It will generate the correct tensor core index as described in (4.10) to initiate tensor-train matrix multiplication. In addition, the CMOS layer will also perform the non-linear mapping.

Note that buffers are designed to separate resistive networks between layer 1 and layer 2. The last layer of CMOS contains read-out circuits for RRAM-crossbar and performs logic control for neural network synchronization.

Mapping flow: Fig. 4.8 shows the working flow for the tensor-train based neural network mapping on the proposed architecture. Firstly, the algorithm optimization targeting for the specific application is performed. The neural network compression is performed through layer-wise training process. Then, the design space between compression rate, bit-width and accuracy is explored to determine the optimal neural network configuration (such as number of layers and activation function). Secondly, the architecture level optimization is performed. The RRAM storage on Layer 1 and the

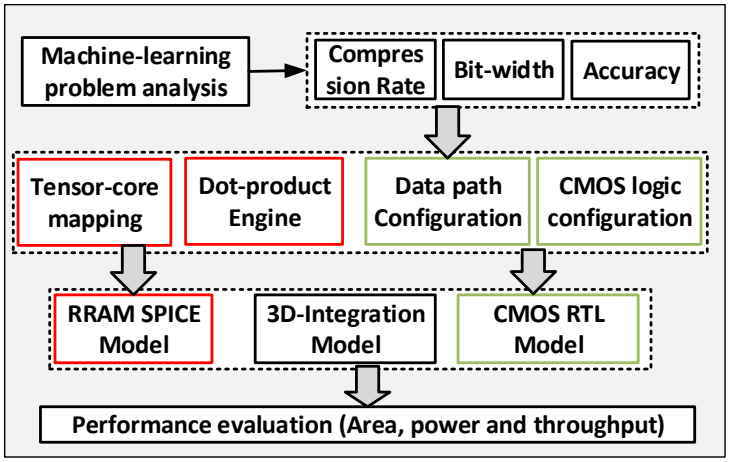


Figure 4.8: Mapping flow for tensor-train based neural network (TNN) on the proposed architecture

computing elements on Layer 2 are designed to minimize the read access latency and power consumption. Furthermore, the CMOS logic is designed based on finite state machine for neural network synchronization. Finally, the whole system is evaluated based on the RRAM SPICE model, CMOS RTL Verilog model and 3D-integration model to determine the system performance.

4.3.2 TNN Accelerator Design on 3D CMOS-RRAM Architecture

In this section, we further discuss how to utilize the proposed 3D multi-layer CMOS-RRAM architecture to design the TNN accelerator. We first discuss the CMOS layer design, which performs the high level control of TNN computation. Then a highly-parallel RRAM based accelerator is introduced with the TNN architecture and dot-product engine.

CMOS Layer Accelerator

To fully map TNN on the proposed 3D multi-layer CMOS-RRAM accelerator, the CMOS logic is designed mainly for logic control and synchronization using top-level state machine. It prepares the input data for computing cores, monitors the states of RRAM logic computation and determines the computation layer of neural network. Fig. 4.9 shows the detailed mapping of the tensorized neural network (TNN) on the proposed 3D multi-layer CMOS-RRAM accelerator. This is a folded architecture by utilizing the sequential operation of each layer on the neural network. The inference data will be collected from RRAM memory through TSV and then sent into vector core to perform vector-matrix multiplication through highly parallel processing elements in the RRAM

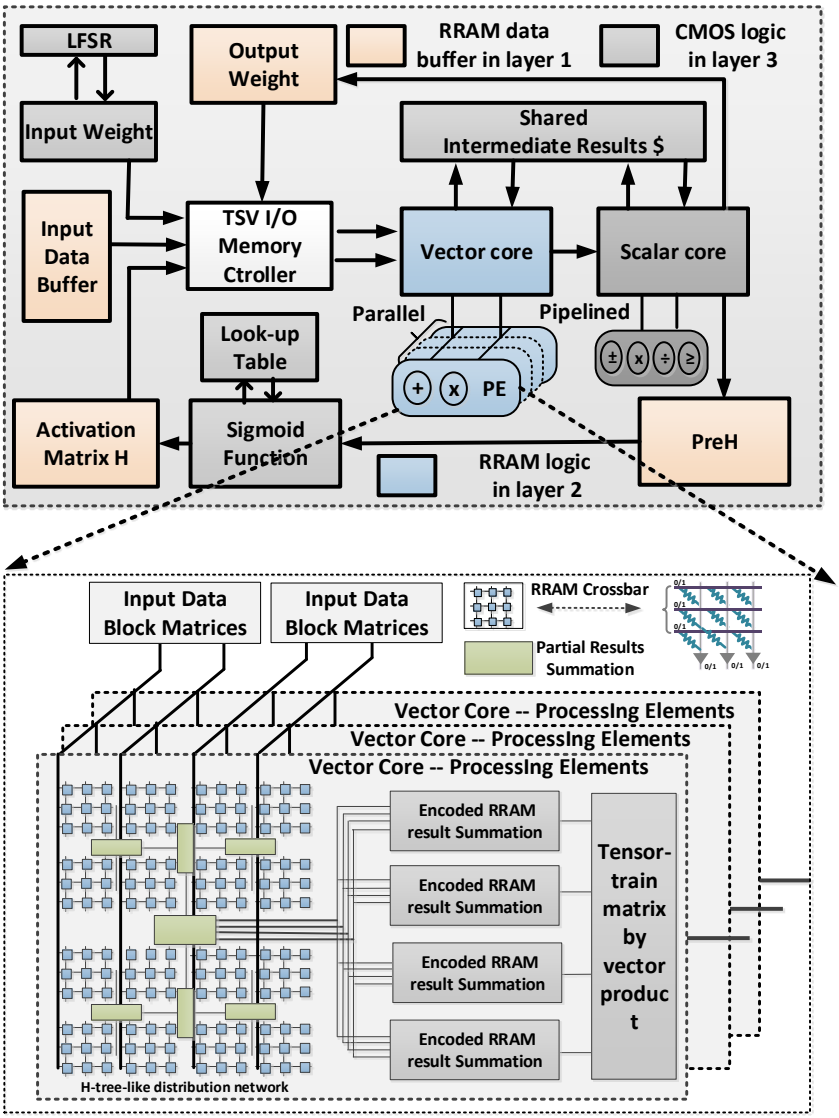


Figure 4.9: Data control and synchronization on layer of CMOS with highly-parallel RRAM based processing elements

layer. The RRAM layer has many distributed RRAM-crossbar structures to perform multiplication in parallel. Then the computed output from RRAM will be transfered to scalar score to perform accumulations. The scaler core can perform addition, subtraction and comparisons. Then the output from the scalar core will be sent to the sigmoid function model for activation matrix in a pipelined fashion, which performs the computation of (4.5). The activation matrix $\boldsymbol{H}$ will be used for the next layer computation. As a result, the whole TNN inference process can be mapped to the proposed 3D multi-layer CMOS-RRAM accelerator.

In addition, to support TNN on RRAM computation, a dedicated index look-up table is formed. Since the weight matrix is actually folded into a high dimensional tensor as

shown in Fig. 4.1, a correct index selection function called bijective function is designed. The bijective function for weight matrix index is also performed by the CMOS layer. Based on the top state diagram, it will choose the correct slice of tensor core $\boldsymbol{G}_i[i, j]$ by determining the i, j index. Then the correct RRAM-crossbar area will be activated to perform vector-matrix multiplication.

RRAM Layer Accelerator

In the RRAM layer, we design the RRAM layer accelerator for highly-parallel computation using single instruction multiple data (SIMD) method to support data parallelism. The discussion of RRAM-crossbar is already introduced in Chapter 3.3.2.

Highly-parallel TNN Accelerator on the RRAM Layer : The TNN accelerator is designed to support highly parallel tensor-train-matrix-by-vector multiplication by utilizing the associative principle of matrix product. According to (4.10), $\boldsymbol{\mathcal{X}}(\boldsymbol{i})$ needs to be multiplied by d matrices unlike the general neural network. As a result, if traditional matrix-vector multiplication in serial is applied, data needs to be stored in the RRAM array for d times, which is time-consuming. Since the size of tensor cores in the TNN is much smaller than the weights in the general neural network, multiple matrix-vector multiplication engines can be placed in the RRAM logic layer. When then input data is loaded, the index of $\boldsymbol{G}_i$ can be known. For example, we need compute $\boldsymbol{X}(\boldsymbol{j})\boldsymbol{G}_1[i_1, j_1]\boldsymbol{G}_2[i_2, j_1]\boldsymbol{G}_3[i_3, j_1]\boldsymbol{G}_i[i_4, j_1]$ given $d = 4$ for the summation in (4.10). $\boldsymbol{G}_1[i_1, j_1]\boldsymbol{G}_2[i_2, j_1]$ and $\boldsymbol{G}_3[i_3, j_1]\boldsymbol{G}_i[i_4, j_1]$ in (4.10) can be pre-computed in a parallel fashion before the input data $\boldsymbol{\mathcal{X}}(\boldsymbol{i})$ is loaded.

As shown in Fig. 4.10, the tensor cores (TC1-6) are stored in the RRAM logic layer. When the input data $\boldsymbol{\mathcal{X}}(\boldsymbol{j})$ comes, the index of each tensor core is loaded by the logic layer controllers first. The controller will write the corresponding data from the tensor cores to RRAM cells. As a result, the matrix-vector multiplication of $\boldsymbol{G}_i$ can be performed in parallel to calculate the intermediate matrices while $\boldsymbol{\mathcal{X}}(\boldsymbol{i})$ is in the loading process. After all the intermediate matrices are obtain, they can be multiplied by $\boldsymbol{\mathcal{X}}(\boldsymbol{i})$ so that the operations in RRAM logic layer can be efficient.

Highly-parallel Dot-product Engine on the RRAM Layer: We further develop the digitalized RRAM based dot-product engine on the RRAM layer. The tensor-train-matrix-by-vector operation can be efficiently accelerated by the fast dot-product engine on the RRAM layer, as shown in Fig. 4.10. By taking correct index of cores, each operation can be divided into a vector-vector dot product operation. Here, we design the dot-product engine based on [122]. We use the output matrix $\boldsymbol{Y}$, input matrices $\boldsymbol{X}$ and $\boldsymbol{\Phi}$ for better understanding. The overall equation is $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\Phi}$ with $\boldsymbol{Y} \in \mathbb{R}^{M\times m}$, $\boldsymbol{X} \in \mathbb{R}^{M\times N}$

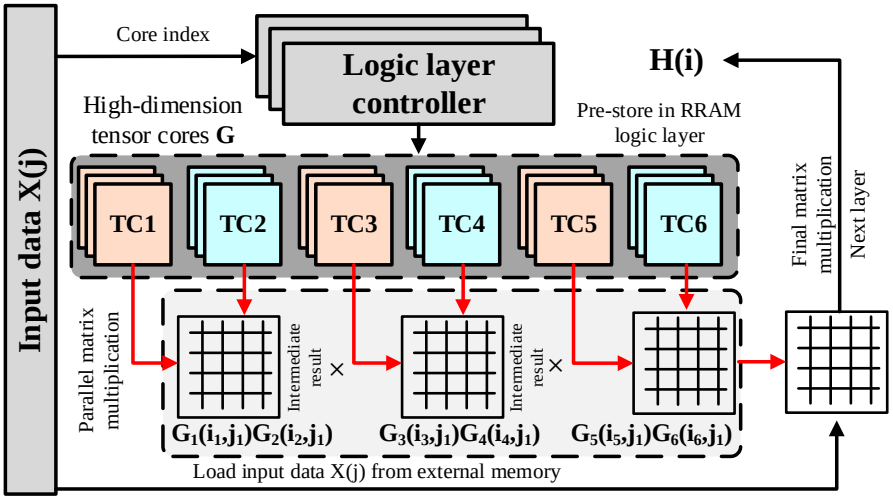


Figure 4.10: RRAM based TNN accelerator for highly parallel computation on tensor cores

and $\mathbf{\Phi} \in \mathbb{R}^{N\times m}$. For every element in $\boldsymbol{Y}$, it follows

$$y_{ij} = \sum_{k=1}^{N} x_{ik}\varphi_{kj}, \tag{4.20}$$

where x and φ are the elements in $\boldsymbol{X}$ and $\mathbf{\Phi}$ respectively. The basic idea of the implementation is to split the matrix-vector multiplication to multiple inner-product operations of two vectors. Such multiplication can be expressed in binary multiplication by adopting fixed point representation of x_{ik} and φ_{kj}:

$$\begin{aligned} y_{ij} &= \sum_{k=1}^{N}(\sum_{e=0}^{E-1} B_e^{h_{ik}} 2^e)(\sum_{g=0}^{G-1} B_g^{\gamma_{kj}} 2^g), \\ &= \sum_{e=0}^{E-1}\sum_{g=0}^{G-1}(\sum_{k=1}^{N} B_e^{h_{ik}} B_g^{\gamma_{kj}} 2^{e+g}) = \sum_{e=0}^{E-1}\sum_{g=0}^{G-1} s_{eg} 2^{e+g} \end{aligned} \tag{4.21}$$

where s_{eg} is the accelerated result from RRAM-crossbar. $B^{h_{ik}}$ is the binary bit of h_{ik} with E bit-width and $B^{\gamma_{kj}}$ is the binary bit of γ_{kj} with G bit-width. As mentioned above, bit-width E and G are decided during the algorithm level optimization. The matrix-vector multiplication based on (4.21) can be summarized in four steps on the RRAM layer.

Step 1: Index Bijection: Select the correct slice of tensor cores $\boldsymbol{G}_d[i_d, j_d] \in \mathbb{R}^{r_d\times r_{d+1}}$, where a pair of $[i_d, j_d]$ determines a slice from $\boldsymbol{G}_d \in \mathbb{R}^{r_d\times n_d\times r_{d+1}}$. In our current example, we use $\boldsymbol{X} \in \mathbb{R}^{M\times N}$ and $\mathbf{\Phi} \in \mathbb{R}^{N\times m}$ to represent two selected slices from cores $\mathbf{G}_1$ and $\mathbf{G}_2$.

Step 2: Parallel Digitization: The matrix multiplication $\boldsymbol{X} \times \mathbf{\Phi}$ requires Mm times

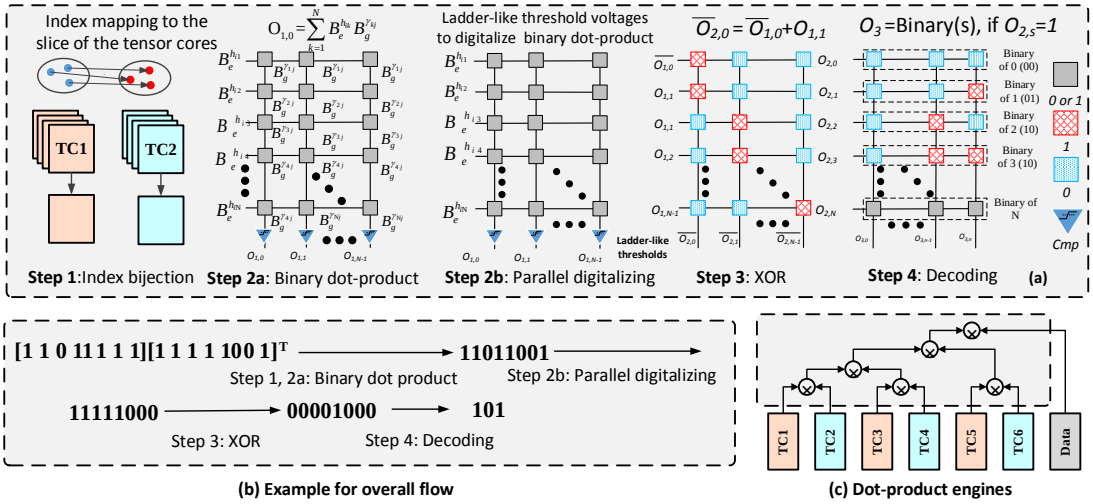


Figure 4.11: (a) RRAM based dot-product engine (b) an example of dot-product overall flow (c) tree-based parallel tensor cores multiplication

N-length vector dot product multiplication. Therefore, an $N \times N$ RRAM-crossbar is required. For clarity, we explain this steps as two sub-steps but they are implemented on the same RRAM-crossbar.

1. **Binary Dot-Product Process**: Each inner-product is produced by the RRAM-crossbar as shown in Fig. 4.11(a). $B_e^{h_{ik}}$ is set as the crossbar input and $B_g^{\gamma_{kj}}$ is written in RRAM cells. The multiplication process on RRAM follows (3.20).

2. **Digitalizing**: In the $N \times N$ RRAM-crossbar, resistance of RRAMs in each column are the same, but V_{th} among columns are different. As a result, the output of each column is calculated based on ladder-like threshold voltages $V_{th,j}$ for parallel digitalizing.

If the inner-product result is s, the output of step 2 is like $(1...1,0...0)$, where $O_{1,s} = 1$ and $O_{1,s+1} = 0$. Fig. 4.11(b) gives an example of the digitalized output.

Step 3: XOR: It is to identify the index of s with the operation $O_{1,s} \oplus O_{1,s+1}$. Note that $O_{1,s} \oplus O_{1,s+1} = 1$ only when $O_{1,j} = 1$ and $O_{1,j+1} = 0$ from Step 2. The mapping of RRAM-crossbar input and resistance is also shown in 4.10(c), and threshold voltage configuration is $V_{th,j} = \frac{V_r R_s}{2R_{on}}$. Therefore, the index of s is identified by XOR operation.

Step 4: Encoding: the fourth step produces s in the binary format as an encoder with the thresholds from Step 3. The input from the third step produces $(0...1,0...0)$ like result where only the s-th input is 1. As a result, only the s-th row is read out and no current merge occurs in this step. The corresponding binary format $binary(s)$ is an intermediate result and stored in the s-th row. The encoding step needs an $N \times n$

RRAM-crossbar to generate $binary(s)$, where $n = \lceil \log_2 N \rceil$ is the number of bits in order to represent 1 to N in the binary format.

By applying these four steps, we can map different tensor cores on RRAM-crossbars to perform the matrix-vector multiplication in parallel. Compared to the state-of-arts realizations, this approach can perform the matrix-vector multiplication faster and more energy-efficient.

4.4 Experiment Results

4.4.1 TNN Performance Evaluation and Analysis

In this section, we will firstly show our experiment setup and various datasets used for performance evaluation. The performance of shallow TNN is discussed with accuracy and compression. After that, a stacked auto-encoder based deep TNN is elaborated. Then, a fine-tuned TNN with tensor core quantization is presented to compare with the state-of-the-art results. Finally, a 3D multi-layer CMOS-RRAM based hardware implementation for TNN is discussed and compared to other computing platforms.

Experiment Setup

The neural network design is performed on Matlab using Tensor-train toolbox [72, 118] and Matcovnet [123]. All the experiments are performed on the Dell server with 3.47GHz Xeon Cores and 48G RAM. Two GPU (Quadro 5000) are also used to accelerate the backward propagation (BP) training process of tensor-train layers [72]. We analyze shallow TNN and deep TNN on UCI dataset [99] and MNIST [124] dataset. To evaluate the model compression, we compare our shallow TNN with SVD based node pruned method [125] and general neural network [82]. We also compare auto-encoder based deep TNN with various other works [72, 73, 126, 70]. The details of each dataset are summarized in Table 4.2.

Shallow TNN

As discussed in Chapter 4.2.1, a shallow TNN is a single hidden layer feed forward neural network. The first layer is a randomly generated tensor-train based input weight and the second layer is optimized by modified alternating least squares method (MALS). In this experiment, we can find that tensor-train based neural network shows a fast inference process with model compressed when the tensor rank is small. To evaluate this, we apply the proposed learning method comparing to general neural network [82] on

Table 4.2: Specification of benchmark datasets

Dataset [99] Name	**Training Samples**	**Inference Sampels**	**Attributes**	**Classes**
Iris	120	30	3	4
Adult	24580	6145	14	2
Credict	552	138	14	2
Diabets	154	612	8	2
Glass	171	43	10	7
Leukemia	1426	5704	38	2
Liver	276	70	16	2
Segment	1848	462	19	12
Wine	142	36	3	12
Mushroom	6499	1625	22	3
Vowel	422	106	10	11
Shuttle	11600	2900	9	7
CIFAR-10 [127]	50000	10000	32x32x3	10
MNIST [124]	60000	10000	28x28	10
MSRC-12‡ [128]	2916	2965	1892	12
MSR-Action3D‡ [129]	341	309	672	20

‡ Extracted features from action sequences

Table 4.3: Performance comparison between tensorized neural network (TNN), general neural network (NN) and SVD pruned neural netowrk (SVD) on UCI dataset

Dataset§	**iris**	**adult**	**credict**	**diabetes**	**Glass**	**leukemia**	**liver**	**segment**	**shuttle**
Model†	128x4x3	128x14x2	128x14x2	128x8x2	64x10x7	256x38x2	128x16x2	128x19x12	1024x9x7
TNN‡ **Inf.-time (s)**	7.57E-04	8.70E-03	9.26E-04	8.07E-04	6.44E-04	4.97E-04	6.15E-04	5.72E-03	5.29E-02
TNN‡ **Inf.-Acc**	0.968	0.788	0.778	0.71	0.886	0.889	0.714	0.873	0.995
NN Inf.-time (s)	1.14E-03	1.04E-02	2.20E-03	1.80E-03	1.50E-03	1.50E-03	2.00E-03	1.84E-02	5.41E-02
NN Inf.-Acc	0.991	0.784	0.798	0.684	0.909	0.889	0.685	0.886	0.989
SVD Inf.-time (s)	8.73E-04	9.36E-03	2.02E-03	1.70E-03	1.12E-03	1.49E-03	1.63E-03	1.52E-02	3.81E-02
SVD Inf.-Acc	0.935	0.783	0.743	0.496	0.801	0.778	0.7	0.847	0.986
TNN Comp.	1.12	1.8686	1.7655	3.1373	1.3196	1.8156	1.5802	2.6821	1.5981
SVD Comp.	1.113	1.113	1.113	1.113	1.103	1.113	1.113	1.113	1.111
TNN Acc. Lss	0.023	-0.004	0.02	-0.026	0.023	0	-0.029	0.013	-0.006
SVD Acc. loss	0.056	0.001	0.055	0.188	0.108	0.111	-0.015	0.039	0.003
TNN Speed-up	1.506	1.195	2.376	2.23	2.329	3.018	3.252	3.207	1.023
SVD Speed-up	1.306	1.111	1.087	1.061	1.343	1.008	1.228	1.211	1.419

† $L \times n \times m$, where L is number of hidden nodes, n is the number of features and m is the number of classes. All the datasets are applied to single hidden layer neural network with L sweep from 64 to 1024.
§ Detailed information on dataset can be found from [99]. We randomly choose 80% of total data for training and 20% for inference.

‡ Rank is initialized to be 2 for all tensor cores.

UCI dataset. Please note that the memory required for TNN is $\sum_{k=1}^{d} n_k r_{k-1} r_k$ comparing to $N = n_1 \times n_2 \times ... \times n_d$ and the computation process can be speed-up from $O(NL)$ to $O(dr^2 n_m \ max(N,L))$, where n_m is the maximum mode size of the tensor-train. Table 4.3 shows detailed comparison of speed-up, compressed-model and accuracy between TNN, general neural network and SVD pruned neural network. It clearly shows that

proposed method can accelerate the inference process comparing to general neural network. In addition, our proposed method only suffers around 2% accuracy loss but SVD based method has different losses (up to 18.8 %). Furthermore, by tuning the tensor rank we can achieve 3.14× compression for diabetes UCI dataset. Since we apply 10% node pruning by removing the smallest singular values, the model compression remains almost the same for different benchmarks.

As discussed in Chapter 4.2.6, bit-width configuration is an effective method to reduce the model size. To achieve such goal, we apply non-uniform quantization for the tensor core weights. As shown in Fig. 4.12, the probability density function of tensor core weights can be modeled as Gaussian distribution. For such known pdf, we can effectively find the optimal level representative values with minimized mean square error. Fig 4.13 shows the trade-off between accuracy, bit-width and compression rate on MNIST dataset with shallow tensorized neural network.

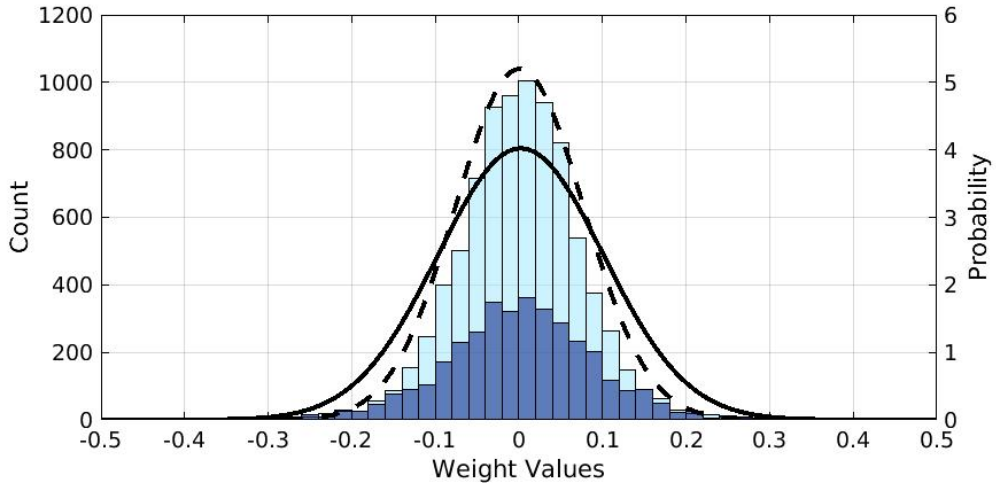


Figure 4.12: Tensorized neural network layer 1 weight and layer 2 weight histogram with approximated Gaussian distribution

Deep TNN

For a deep tensorized neural network, we mainly investigate auto-encoder based multi-layer neural network on MNIST dataset. We firstly discuss the learnt filters by the proposed auto-encoder. Then the hyper-parameter of TNN such as tensor-train ranks and number of hidden nodes are discussed with respect to the inference time, neural network compression rate and accuracy.

Fig. 4.14 shows the first layer filter weights of proposed auto-encoder. Please note that the TNN architecture for MNIST is $\boldsymbol{W}_1$ (784×1024), $\boldsymbol{W}_2$ (1024×1024) and

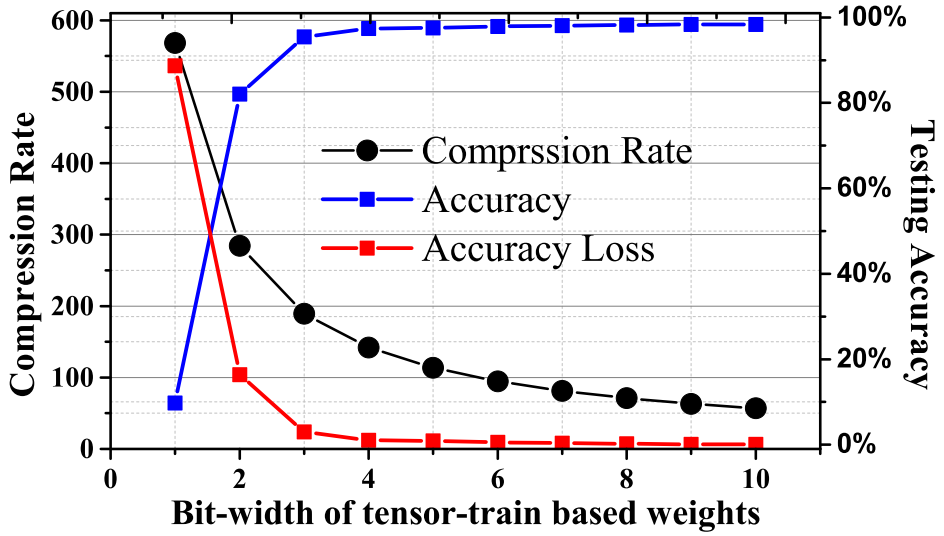
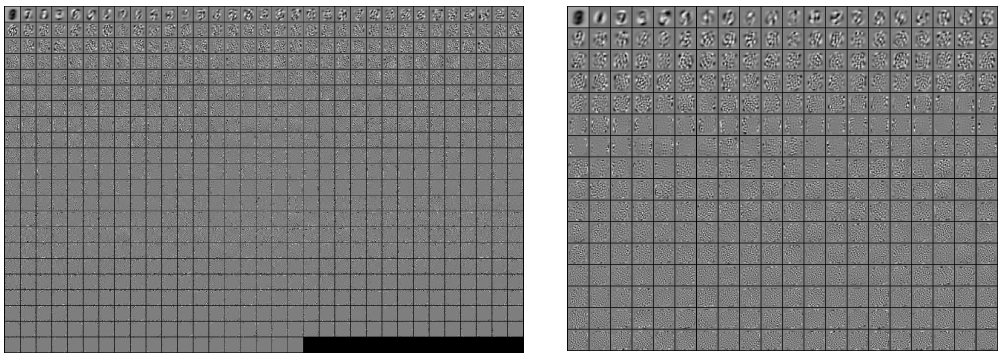


Figure 4.13: Compression rate and accuracy with increasing bit-width of tensor core weights



(a) (b)

Figure 4.14: Visualization of layer-wise learned weights on layer 1 by reshaping each independent column into square matrix with (a) rank = 50 and (b) rank = 10 on MNIST dataset (The range of weights is scaled to [0 1] and mapped to grey-colored map, where 0 is mapped to black and 1 to white)

$\boldsymbol{W}_3$ (1024×10). We re-arrange each column into a suqare image and visualize on each cell of the visualization panel. We only visualize those independent columns. Therefore, from Fig. 4.14, we can see that the larger ranks, the more independent columns and the more visualization cells. We can also find that in Fig. 4.14(a), large tensor ranks can learn more filters. The first three rows are mainly the low frequency information and then the middle rows consist of a little detailed information of the input images. In the last few rows, we can see more sparse cells there representing high frequency information. In comparisons, Fig. 4.14(b) shows less independent filters due to the smaller tensor rank. However, we still can find similar filter patterns in it. It is a subset of Fig.

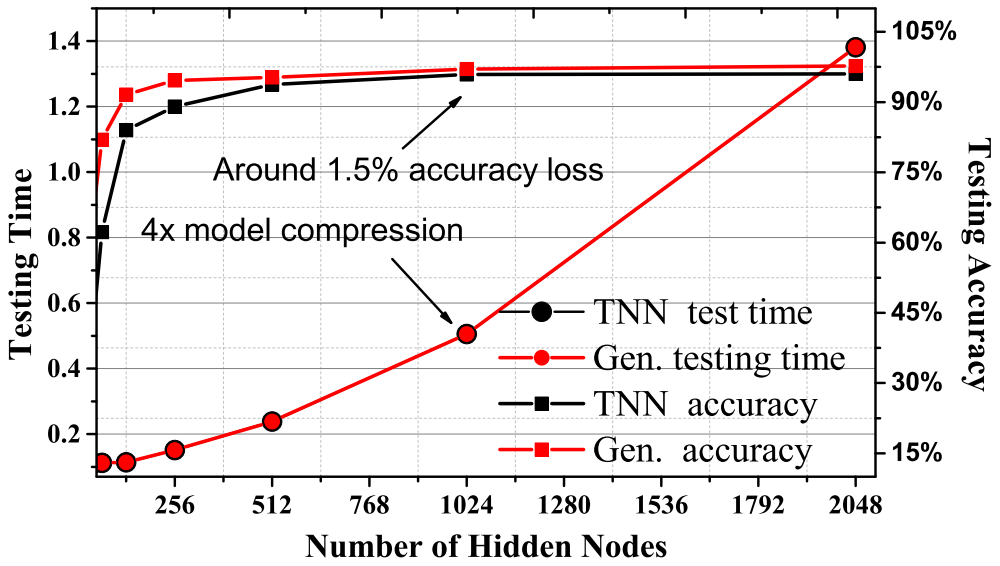


Figure 4.15: Inference time and accuracy comparison between TNN and general neural network (Gen.) with varying number of hidden nodes

4.14(a) filter results. Therefore, by reducing ranks, we can effectively tune the number of filters required, which will provide the freedom of finding an optimal number of filters to save parameters.

Fig. 4.15 shows the inference accuracy and running time comparisons for MNIST dataset. It shows a clear trend of accuracy improvement with increasing number of hidden nodes. The running time between TNN and general NN is almost the same. This is due to the relative large rank $r = 50$ and computation cost of $O(dr^2 n_m \ max(N,L))$. Such tensor-train based neural network achieve $4\times$ compression within 1.5% accuracy loss under 1024 number of hidden nodes respectively. Details on model compression are shown in Table 4.4. From Table 4.4, we can observe that the compression rate is directly connected with the rank r, where the memory storage can be simplified as dnr^2 from $\sum_{k=1}^{d} n_k r_{k-1} r_k$ but not directly link to the number of hidden nodes. We also observe that by setting tensor core rank to 35, $14.85\times$ model compression can be achieved with acceptable accuracy loss. The compression rate can be further improved using quantized TNN to $59.4\times$ compression rate. Table 4.4 also shows the clear effect of quantization on the compression rate. In generally, bit-width quantization can help improve $3\times$ more compression rate on neural network. Therefore, low rank and quantized tensor cores are important and orthogonal methods to increase compression rate.

Fig. 4.16 shows the increasing accuracy when we increase the rank of tensor-train based weights. Here, we have two factorization of the input image 28×28, which we refer to TNN 1 ($2 \times 2 \times 2 \times 2 \times 7 \times 7$) and TNN 2 ($4 \times 4 \times 7 \times 7$). We can observe that changing the weight factorization will slightly affect the accuracy of the neural network

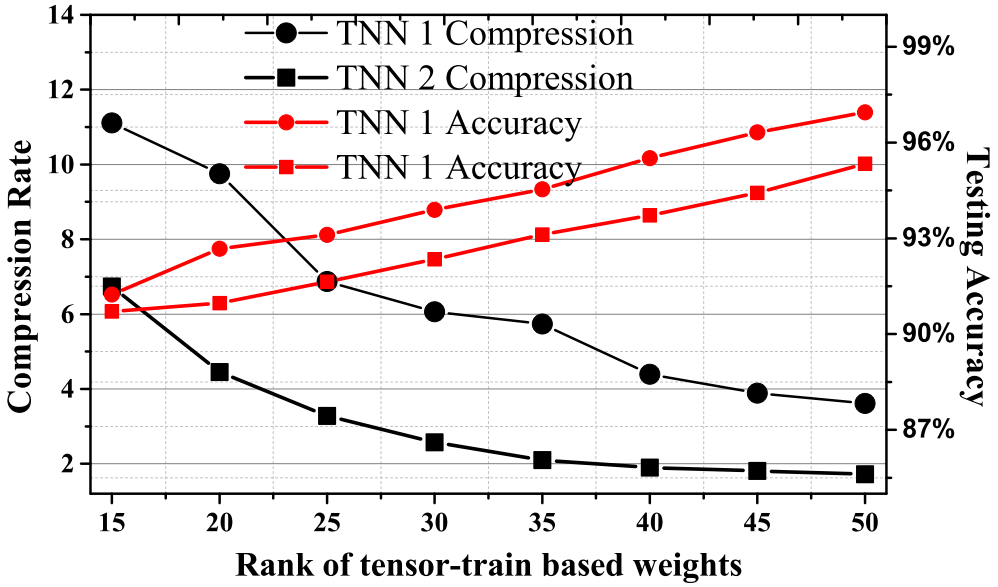


Figure 4.16: Compression and accuracy comparison between two factorization (TNN 1 ($2 \times 2 \times 2 \times 2 \times 7 \times 7$) and TNN 2 ($4 \times 4 \times 7 \times 7$) of tensorized weights with varying ranks

Table 4.4: Model Compression under different number of hidden nodes and tensor ranks on MNIST dataset

No Hid†	32	64	128	256	512	1024	2048
Compression	5.50	4.76	3.74	3.23	3.01	4.00	8.18
Rank‡	15	20	25	30	35	40	45
Compression	25.19	22.51	20.34	17.59	14.85	12.86	8.63
Accuracy (%)	90.42	90.56	91.41	91.67	93.47	93.32	93.86

‡ Number of hidden nodes are all fixed to 2048 with 4 fully connected layers.

† All tensor Rank is initialized to 50.

by around 1% accuracy. Furthermore, we find the change trend of compression rate is the same for both factorization modes but TNN 1 can compress more parameters. We can conclude that decomposing weights to more tensor cores will empirically improve the compression rate as well as accuracy.

Fine-tunned TNN

The proposed TNN can also perform end-to-end learning to fine-tune the compressed model to improve the accuracy result. Under 1024 number of hidden nodes and 15 maximum rank, we can achieve 91.24 % inference accuracy. Then we perform left-to-right sweep to remove small singular values to make the compression rate 64. This compression rate is set mainly for the result comparisons with other works. After fixing the compression rate, the fine-tuning process is shown as Fig. 4.17. The *top1err* means

Table 4.5: Inference-error comparison with 64× model compression under single hidden layer neural network

Method	Error rates
Random Edge Removal [126]	15.03%
Low Rank Decomposition [73]	28.99%
Distilled Knowledge in neural network [70]	6.32%
Hashing trick based compression [64]	2.79%
Tensorzing Neural Network [72]	1.90%
Layer-wise learning of TNN	2.21%
Quantized Layer-wise learning o TNN ‡	1.59%

‡ Only tensorized layers are quantized with 9 bit-width.

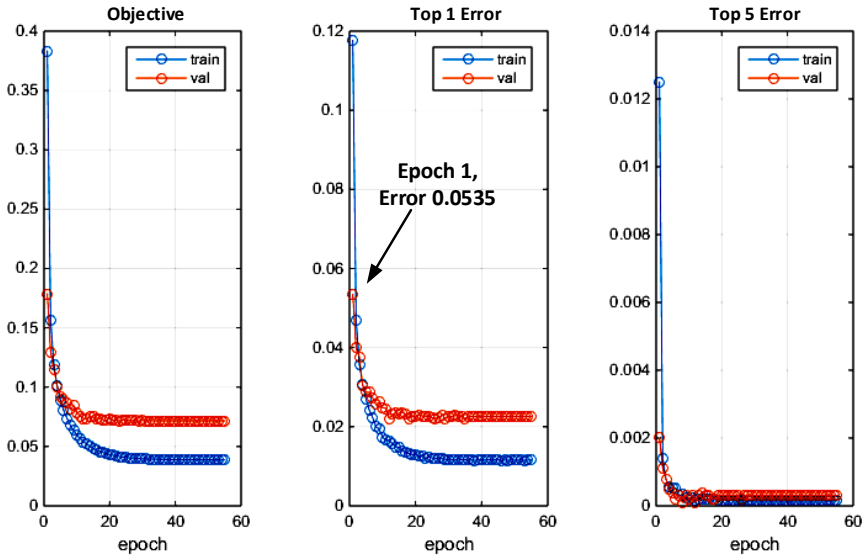


Figure 4.17: Fine tuning process from proposed TNN layer-wise training method with 64× compression on single hidden layer neural network on MNIST dataset

the prediction accuracy after one guess and *top5err* refers to the prediction accuracy after five guesses. We can find a very deep accuracy improvement at the first 5 epochs and then it becomes flat after 20 epochs. The final error rate is 2.21%. By adopting non-uniform quantization of 9 bit-width on TNN with fixed 64 compression rate, we achieve a even lower error rate (1.59%). We also compare this with other works as summarized in Table 4.5.

To have a fair comparison with [8], we also adopt the same network, LeNet-300-100 network [124], which is a four-layer fully-connected neural network (784×300, 300×100 , 100×10). Under such configuration, [8] can achieve 40× compression with error rate 1.58 % using quantization (6 bit-width precision), pruning and huffman coding techniques. In our proposed TNN, with the same compression rate 40×, we can achieve

1.63 % error rate under single floating point (32 bit) precision. By adopting 9 bit-width on tensorized layer, we can achieve smaller error rate 1.55 % with the same compression rate [4]. Moreover, the proposed TNN provides more flexibility with the simplified compression process. The high compression rate from [8] is a state-of-the-art result. However, this is achieved using 3 techniques, and they are: pruning, quantization and Huffman coding as mentioned above. The pruning method is to remove small-weight connections where all connections with weights below a threshold are removed. Thereafter the pruned neural network is re-trained. Note that the threshold is determined by the trial-and-error method, which requires multiple trainings before arriving at an optimal threshold. The second method is quantization. Again, the bit-width is determined by the trial-and-error method with backwards propagation to re-train the neural network. Finally, the Huffman coding is a lossless data compression technique to further compress the neural network. Our proposed method is using the layer-wise training approach to search for the optimal ranks of the tensor cores, which save time in finding the optimal ranks. Moreover, the quantization of the weights is performed less aggressively. As such, the re-training process of quantized neural network becomes optional, which provides more flexibility between training time, accuracy and compression. Moreover, Huffman coding can also be applied to our tensorized neural network. Therefore, this method offers more flexibility.

Therefore, we can conclude that using a tensor-train layer on the fully-connected layers provide more flexible trade-off between network compression and accuracy. An end-to-end fine-tuning can further improve the accuracy without compromising compression rate.

4.4.2 TNN Benchmarked Result

In this section, we will further discuss two applications. One is object recognition using deep convolution neural network on CIFAR-10 [127] dataset. The other one is human action recognitions on MSRC-12 Kinect and MSR-Action3D dataset [128, 129] . The main objective here is to achieve state-of-the-art performances with significant neural network compression.

Object Recognition

Here, we discuss object recognition with convolution neural network (CNN) on CIFAR-10 dataset. CIFAR-10 dataset consists of $60,000$ images with size of $32 \times 32 \times 3$ under

[4]The improvement of accuracy is mainly due to the increased rank value since both tensor-train and quantization techniques are applied to maintain $64\times$ compression rate

10 different classes. This dataset contains 50,000 training images and 10,000 inference images.

For CNN architecture, we adopt LeNet-alike neural network [124] as shown in Table 4.6. We use a two-layer fully-connected TNN to replace the original fully-connected layers. Therefore, the 512 neural output will be treated as input features and we built two tensorized layers of $512 \times N$ and $N \times 10$, where N represents number of hidden nodes. For $N = 512$, our proposed TNN has 12416 (7296+5120) number of parameters of fully-connected layers, which is $5.738\times$ and $1.752\times$ for fully-connected layers and the whole neural network. The inference accuracy is 75.62% with 3.82% loss comparing to original network. For $N = 1024$, our proposed TNN can perform the compression of $4.045\times$ and $1.752\times$ for fully-connected layers and the whole neural network respectively. The inference accuracy is 77.67% with 1.77% loss comparing to original network. This is slightly better than [72] in terms of accuracy, which is $1.7\times$ compression of the whole network with 75.61 % accuracy. Another work [73] can achieve around $4\times$ compression on fully-connected layers with around 74% accuracy and 2 % accuracy loss. Note that the purpose of the proposed tensorized neural network is to compress the neural network with a negligible accuracy loss. The LeNet neural network architecture achieves accuracy 79.44% and we use this architecture to compare with other existing works [72, 73]. By adopting non-uniform tensor core quantization (6 bit-width), we can achieve $21.57\times$ and $2.19\times$ compression on fully-connected layers and total neural network respectively with 77.24 % performance (2.2% accuracy loss). Therefore, our proposed TNN can achieve high compression rate with maintained accuracy.

Table 4.6: CNN architecture parameters and compressed fully-connected layers

Layer	Type	No. of maps and neurons	Kernel	Sride	Pad	No. Param.	Compr. Param.
0	Input	3 Maps of 32x32 neurons	—	—	—	—	—
1	Convolutional	32 Maps of 32x32 neurons	5x5	1	2	2432	2432
2	Max Pooling	32 Maps of 16x16 neurons	3x3	2	[0 1 0 1]	—	—
3	ReLu	32 Maps of 16x16 neurons	—	—	—	—	—
4	Convolutional	32 Maps of 16x16 neurons	5x5	1	2	25632	25632
5	Relu	32 Maps of 16x16 neurons	—	—	—	—	—
6	Ave. Pooling	32 Maps of 8x8 neurons	3x3	2	[0 1 0 1]	—	—
7	Convolutional	32 Maps of 8x8 neurons	5x5	1	2	25632	25632
8	Ave. Pooling	32 Maps of 8x8 neurons	3x3	2	[0 1 0 1]	—	—
9	Reshape	512 Maps of 1x1 neurons	–	—	—	—	—
10	Fully-Connected	64 Maps of 1x1 neurons	1x1	1	0	32832	7360 (4.40×)
11	Relu	64 Maps of 1x1 neurons	–	—	—	—	—
12	Fully-Connected	512 Maps of 1x1 neurons	1x1	1	0	33280	10250 (3.747×)
13	Fully-Connected	10 Maps of 1x1 neurons	1x1	1	0	5130	

Table 4.7: Gesture classes and the number of annotated instances for each class in MSRC-12 Kinect dataset

Gestures	Number of Insts.	Gestures	Number of Insts.
Start System	508	Duck	500
Push Right	522	Goggles	508
Wind it up	649	Shoot	511
Bow	507	Throw	515
Had Enough	508	Change Wepon	498
Beat Both	516	Kick	502

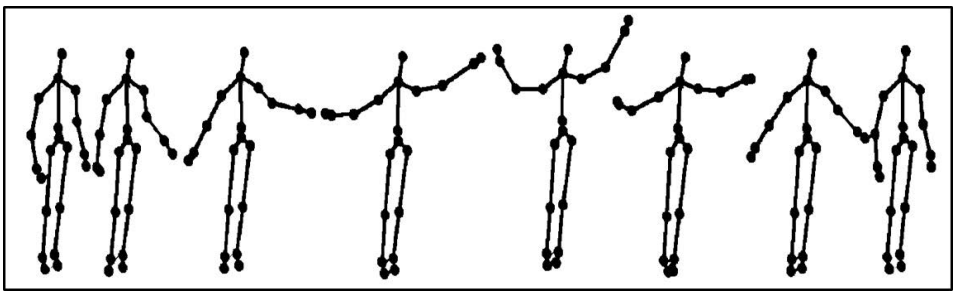


Figure 4.18: Human action from MSRC-12 Kinect dataset: a sequence of 8 frames action for the Start System Gesture [2].

Human Action Recognition

MSRC-12 is a relatively large dataset for action/gesture recognition from 3D skeleton data [128, 2] recorded from Kinect sensors. The dataset has 594 sequences contianing the performances of 12 gestures by 30 subjects with 6244 annotated gestures. The summary of 12 gestures and number of instances are shown in Table 4.7.

In this experiment, we adopt the standard experiment configuration by splitting half of the persons for training and half for inference. We use the covariances of 3D joints description as feature extractor. As shown in Fig. 4.18, the body can be represented by K points, where $K = 20$ for MSRC-12 dataset. The body action can be performed in T frames and we can set x, y and z representing coordinates of the i-th joint at frame t. The sequence of joint location can be represented as $\boldsymbol{S} = [x_1, x_2, ..., x_K, y_1, y_2, ..., y_K, z_1, z_2, ...z_K]$. Therefore, the covariance of the sequence is:

$$C(\boldsymbol{S}) = \frac{1}{T-1} \sum_{t=1}^{T} (\boldsymbol{S} - \tilde{\boldsymbol{S}})(\boldsymbol{S} - \tilde{\boldsymbol{S}})^T \tag{4.22}$$

where $\tilde{\boldsymbol{S}}$ is the sample mean. Since covariance matrix is diagonal symmetric matrix, we only use the upper triangular. We also add temporal information of the sequence to the features. We follow the same feature extraction process from [2].

Table 4.8: MSRC-12 human action recognition accuracy and comparisons

Method	**Accuracy**
Hierarchical Model on Action Recognition [130]	66.25%
Extended LC-KSVD [131]	90.22%
Temporal Hierarchy Covariance Descriptors [2]	91.70%
Joint Trajectory-CNN [132]	93.12%
Sliding Dictionary-Sparse Representation [133]	92.89%
Proposed tensorized neural network (average)	91.41%
Proposed tensorized neural network (peak)	93.40%

Based on the aforementioned feature extraction, the input feature size of one sequence is 1892. For human action recognitions, we use a four-layer neural network architecture, which is defined as 1892×1024, 1024×1024 and 1024×10 with Sigmoid function. We set the maximum tensors rank to 25 and decompose the input weight into a 8-dimensional tensors with mode size [2 2 11 43] and [2 2 2 128]. The neural network compression rate comparing to general neural network is $8.342\times$ and $28.26\times$ without and with non-uniform quantization respectively.

Fig. 4.19 shows the confusion matrix of proposed TNN method, where the darker the block is, the higher prediction probability it represents. For example, for the class 1 "Start System", the correct prediction accuracy is 82 %. However, as shown in the first row, the most mis-classified class is class 9 and class 11. Both are with 5% probability. The worst case is class 11, which has only 60% accurate prediction probability. The average prediction accuracy is 91.41% after 25 repetitions and the comparison to the existing works is summarized in Table 4.8. We also report our best prediction accuracy 93.4%. Therefore, it clearly shows that our TNN classifier can effectively perform human action recognition close to the state-of-the-art result.

In addition, we have also performed the 3D-action recognition on MSR-Action3D dataset [129]. This dataset has 20 action classes, which consists of a total of twenty types of segmented actions: *high arm wave, horizontal arm wave, hammer, hand catch, forward punch, high throw, draw x, draw tick, draw circle, hand clap, two hand wave, sideboxing, bend, forward kick, side kick, jogging, tennis swing, tennis serve, golf swing, pick up and throw*. Each action starts and ends with a neutral pose and performed by 10 subjects, where each subject performed each action two or three times. We have used 544 sequences, where each sequence is a recoding of the skeleton joint location. Here, we apply the same feature extractors as MSRC-2012 and use our proposed TNN as classifier. We adopt a four-layer neural network, which is 672×1024, 1024×1024 and 1024×20. The compression rate is $3.318\times$ and can be further improved to $11.80\times$ with 9-bit quantization. As shown in Table 4.9 , comparing to other methods, we can

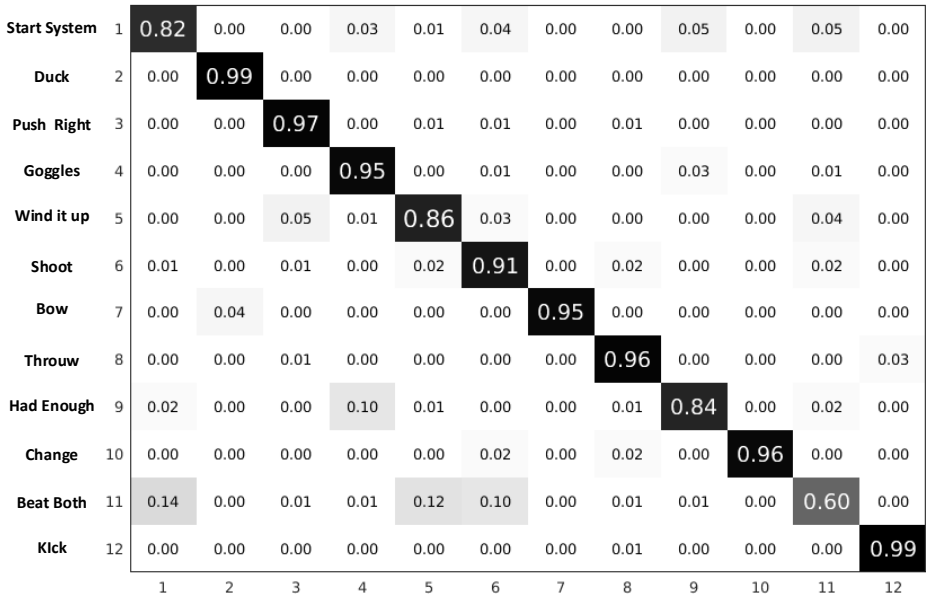


Figure 4.19: Confusion matrix of human action recognitions from MSRC-12 Kinect dataset by 50% random subject split with 25 times repetition.

achieve the state-of-the-art result with 88.95% recognition accuracy.

Table 4.9: MSR-Action3D human action recognition accuracy and comparisons

Method	**Accuracy**
Recurrent Neural Network [134]	42.50%
Hidden Markov Model [135]	78.97%
Random Occupancy Patterm [136]	86.50%
Temporal Hierarchy Covariance Descriptors [2]	90.53%
Rate-Invariant SVM [137]	89.00%
Proposed tensorized neural network	88.95%

4.4.3 TNN Hardware Accelerator Result

Experiment Settings

In the experiment, we have implemented different baselines for performance comparisons. The detail of each baseline is listed below:

Baseline 1: General CPU processor. The general process implementation is based on Matlab with optimized C-program. The computer server is with 6 cores of 3.46GHz and 64.0GB RAM.

Baseline 2: General GPU processor. The general-purpose GPU implementation is

based on the optimized C-program and Matlab parallel computing toolbox with CUDA-enabled Quadro 5000 GPU [138].

Baseline 3: 3D CMOS-ASIC. The 3D CMOS-ASIC implementation with proposed architecture is done by Verilog with $1GHz$ working frequency based on CMOS 65nm low power PDK. Power, area and frequency are evaluated through Synopsys DC compiler (D-2010.03-SP2). Through-silicon via (TSV) area, power and delay are evaluated based on Simulator DESTINY [120] and fine-grained TSV model CACTI-3DD [121]. The buffer size of the top layer is set to $128MB$ to store tensor cores with 256 bits data width. The TSV area is estimated to be 25.0 μm^2 with a capacitance of $21fF$.

Proposed 3D CMOS-RRAM: The settings of CMOS evaluation and TSV model are the same as baseline 3. For the RRAM-crossbar design evaluation, the resistance of RRAM is set as $500K$ and $5M$ as on-state and off-state resistance and 2V SET/RESET voltage according to [139] with working frequency of $200MHz$. The CMOS and RRAM integration is evaluated based on [140].

To evaluate the proposed architecture, we apply UCI [99] and MNIST [141] dataset to analyze the accelerator scalability, model configuration analysis and performance analysis. The model configuration is performed on Matlab first using Tensor-train toolbox [118] before mapping on the 3D CMOS-RRAM architecture. The energy consumption and speed-up are also evaluated. Note that the code for performance comparisons is based on optimized C-Program and deployed as the mex-file in the Matlab environment.

3D Multi-layer CMOS-RRAM Accelerator Scalability Analysis

Since neural network process requires frequent network weights reading, memory read latency optimization configuration is set to generate RRAM memory architecture. By adopting 3D implementation, Simulation results show that memory read and write bandwidth can be significantly improved by 51.53% and 6.51% respectively comparing to 2D implementation as shown in Table 4.10. For smaller number of hidden nodes, read/write bandwidth is still improved but the bottleneck shifts to the latency of memory logic control.

To evaluate the proposed 3D multi-layer CMOS-RRAM architecture, we perform the scalability analysis of energy, delay and area on MNIST dataset [141]. This dataset is applied to multi-layer neural network and the number of hidden nodes may change depending on the accuracy requirement. As a result, the improvement of proposed accelerator with different L from 32 to 2048 is evaluated as shown in Fig. 4.20. With the increasing L, more computing units are designed in 3D CMOS-ASIC and RRAM-crossbar to evaluate the performance. The neural network is defined as a 4-layer network

Table 4.10: Bandwidth improvement under different number of hidden nodes for MNIST dataset

Hidden node†(L)	256	512	1024	2048	4096
Memory required (MB)	1.025	2.55	7.10	22.20	76.41
Memory set (MB)	2M	4M	8M	32M	128M
Write Bandwidth Imp.	1.14%	0.35%	0.60%	3.12%	6.51%
Read Bandwidth Imp.	5.02%	6.07%	9.34%	20.65%	51.53%

†4-layer neural network with 3 full-connected layer $784 \times L$, $L \times L$ and $L \times 10$.

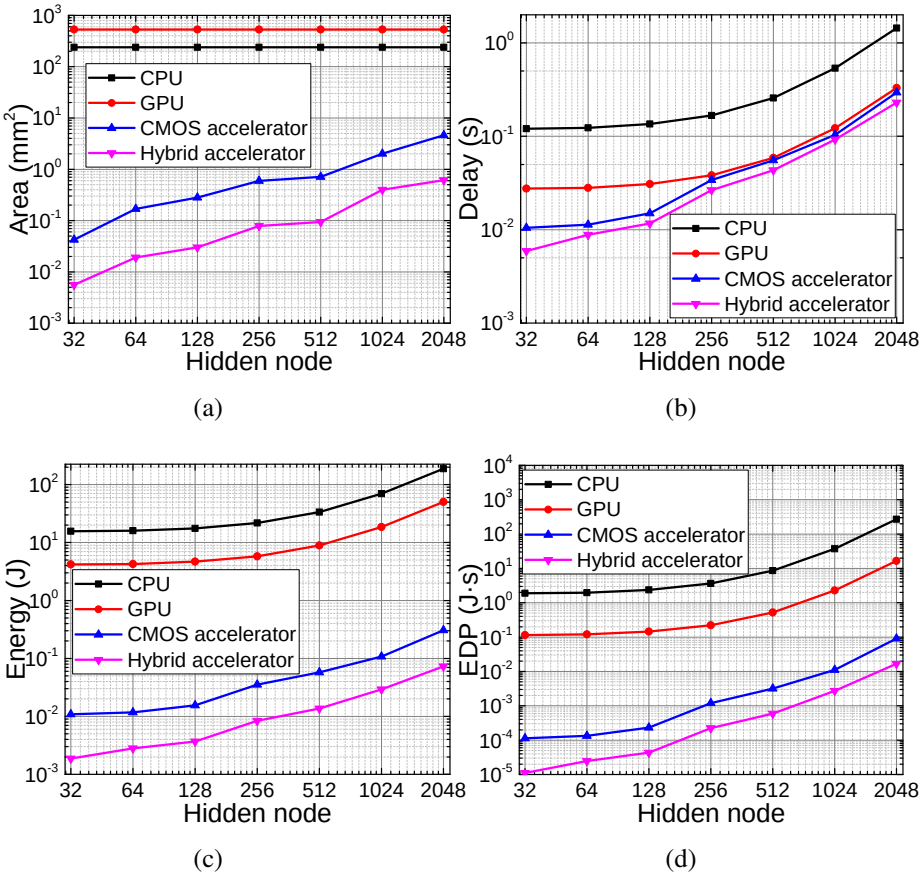


Figure 4.20: Scalability study of hardware performance with different hidden node numbers for: (a) area; (b) delay; (c) energy and (d) energy-delay-product

with weights $784 \times L$, $L \times L$ and $L \times 10$. For computation delay, GPU, 3D CMOS-ASIC and 3D CMOS-RRAM are close when $L = 2048$ according to Fig. 4.20(b). When L reaches 256, 3D CMOS-RRAM can achieve 7.56× area-saving and 3.21× energy-saving compared to 3D CMOS-ASIC. Although the computational complexity is not linearly related to the number of hidden node numbers, both energy consumption and energy-delay-product (EDP) of RRAM-crossbar increase with the rising number of hidden node. According to Fig. 4.20(d), the advantage of the hybrid accelerator becomes

Table 4.11: Inference accuracy of ML techniques under different dataset and bit-width configuration

Datasets	32-bit Acc. (%) & Compr.		4 bit Acc. (%) & Compr.		5 bit Acc. (%) & Compr.		6 bit Acc. (%) & Compr.	
Glass	88.6	1.32	89.22	10.56	83.18	8.45	88.44	7.04
Iris	96.8	1.12	95.29	8.96	96.8	7.17	96.8	5.97
diabets	71	3.14	69.55	25.12	69.4	20.10	71.00	16.75
adult	78.8	1.87	75.46	14.96	78.15	11.97	78.20	9.97
leuke.	88.9	1.82	85.57	14.56	87.38	11.65	88.50	9.71
MNIST	94.38	4.18	91.28	33.44	92.79	26.75	94.08	22.29

smaller when the hidden node increases, but it can still have a 5.49× better EDP compared to the 3D CMOS-ASIC when the hidden node number is 2048.

3D Multi-layer CMOS-RRAM Accelerator Bit-width Configuration Analysis

To implement the whole neural network on the proposed 3D multi-layer CMOS-RRAM accelerator, the precision of real values requires a careful evaluation. Compared to the software double precision floating point format (64-bit), the values are truncated into finite precision. By using the greedy search method, an optimal point for hardware resource (small bit-width) and inference accuracy can be achieved.

Our tensor-train based neural network compression techniques can work with low-precision value techniques to further reduce the data storage. Table 4.11 shows the inference accuracy by adopting different bit-width on UCI datasets [99] and MNIST [141]. It shows that accuracy of classification is not very sensitive to the RRAM configuration bits for UCI dataset. For example, the accuracy of Iris dataset is working well with negligible accuracy at 5 RRAM bit-width. When the RRAM bit-width increased to 6, it performs the same as 32 bit-width configurations. Although the best configuration of quantized model weights varies for different datasets, the general guideline is to start from the 8 bit-width neural network configuration. Based on the report from [142], an 8 bit-width fixed point representation can maintain almost the same performance. We can gradually reduce the bit-width to evaluate the compression rate and accuracy performance. However, to have a bit-width less than 3 would require special training methods and many recent works such as the binary neural network [106] and ternary neural network [143] are all working towards this direction.

Table 4.12: Performance comparison under different hardware implementations on MNIST dataset with 10,000 inference images

Implementation	Power, Freq.	Area (mm^2)	Throughput	Efficiency	Type	Time (s)	Energy(J)
General CPU Processor	130W, 3.46GHz	240, Intel Xeon X5690	74.64 GOPS	0.574 GOPS/W	L1	0.44	57.23
					L2	0.97	125.74
					L3	0.045	5.82
					Overall	1.45	188.8
General GPU Processor	152W, 513MHz	529, Nvidia Quadro 5000	328.41 GOPS	2.160 GOPS/W	L1	0.04	6.05
					L2	0.289	43.78
					L3	0.0024	0.3648
					Overall	0.33	50.19
					Improvement	13.9	1711.8
3D CMOS-ASIC Architecture	1.037W, 1GHz	9.582, 65nm Global Foundary	367.43 GOPS	347.29 GOPS/W	L1	0.032	0.0333
					L2	0.26	0.276
					L3	2.40E-03	2.60E-03
					Overall	0.295	0.312
					Improvement	17.77	7286
3D CMOS-RRAM Architecture	0.371W, 100MHz	1.026, 65nm CMOS and RRAM	475.45 GOPS	1499.83 GOPS/W	L1	0.025	7.90E-03
					L2	0.2	6.40E-01
					L3	1.70E-03	5.00E-04
					Overall	0.23	7.20E-02
					Improvement	6.37	2612

†4-layer neural network with weights 784×2048, 2048×2048 and 2048×10.

3D Multi-layer CMOS-RRAM Accelerator Performance Analysis

In Table 4.12, performance comparisons among C-Program Optimized CPU performance, GPU performance, 3D CMOS-ASIC and 3D multi-layer CMOS-RRAM accelerator are presented for $10,000$ inference images. The acceleration of each layer is also presented for 3 layers (784×2048, 2048×2048 and 2048×10). Please note that the dimension of weight matrices are decomposed into [4 4 7 7] and [4 4 8 8] with 6 bit-width and maximum rank 6. The compression rate is 22.29× and 4.18× with and without bit-truncation. Among the four implementations, 3D multi-layer CMOS-RRAM accelerator performs the best in area, energy and speed. Compared to CPU, it achieves 6.37× speed-up, 2612× energy-saving and 233.92× area-saving. For GPU based implementation, our proposed 3D CMOS-RRAM architecture achieves 1.43× speed-up and 694.68× energy-saving. We also design a 3D CMOS-ASIC implementation with similar structure as 3D multi-layer CMOS-RRAM accelerator with better performance compared to CPU and GPU based implementations. The proposed 3D multi-layer CMOS-RRAM 3D accelerator is 1.283× speed-up, 4.276× energy-saving and 9.339× area-saving compared to 3D CMOS-ASIC.

The throughput and energy efficiency for these four cases are also summarized in Table 4.12. For energy efficiency, our proposed accelerator can achieve 1499.83 GOPS/W, which has 4.30× better energy efficiency comparing to 3D CMOS-ASIC result (347.29 GOPS/W). In comparison to our GPU baseline, it has 694.37× better energy efficiency

comparing to NVIDIA Quadro 5000. For a newer GPU device (NVIDIA Tesla K40), which can achieve 1092 GFLOPS and consume 235W [138], our proposed accelerator has 347.49× energy efficiency improvement.

4.5 Conclusion

This Chapter introduces a tensorized formulation for compressing neural network during training. By reshaping neural network weight matrices into high dimensional tensors with low-rank decomposition, significant neural network compression can be achieved with maintained accuracy. A layer-wise training algorithm of tensorized multilayer neural network is further introduced by modified alternating least-squares (MALS) method. The proposed TNN algorithm can provide state-of-the-arts results on various benchmarks with significant neural network compression rate. The accuracy can be further improved by fine-tuning with backward propagation (BP). For MNIST benchmark, TNN shows 64× compression rate without accuracy drop. For CIFAR-10 benchmark, TNN shows that compression of 21.57× compression rate for fully-connected layers with 2.2% accuracy drop.

In addition, a 3D multi-layer CMOS-RRAM accelerator for highly-parallel yet energy-efficient machine learning is proposed. A tensor-train based tensorization is developed to represent dense weight matrices with significant compression. The neural network processing is mapped on a 3D architecture with high-bandwidth TSVs, where the first RRAM layer is to buffer input data; the second RRAM layer is to perform intensive matrix-vector multiplication using digitized RRAM; and the third CMOS layer is to coordinate the reaming control and computation. Simulation results using the benchmark MNIST show that the proposed accelerator has 1.283× speed-up, 4.276× energy-saving and 9.339× area-saving compared to 3D CMOS-ASIC implementation; and 6.37× speed-up and 2612× energy-saving compared to 2D CPU implementation.

Chapter 5

Distributed-Solver for Networked Neural Network

5.1 Introduction

Distributed machine learning refers to machine learning algorithms designed to work on multi-node computing systems for better performance, accuracy and larger data scale [1]. Increasing the model size as well as the training data size will significantly reduce the learning error and can often improve the system performance. However, this leads to a high computational complexity, which exceeds the capability of single computational node. This is especially true for IoT system. Moreover, the conventional centralized approach suffers the scalability problem since more IoT devices will be deployed with more data collected. Distributed computation is known to provide better scalability. As such, a distributed machine learning algorithm is greatly required to tackle the scalability problem and make use of resource-constrained IoT devices.

To perform machine learning algorithms in a distributed system such as IoT based smart buildings as shown in Fig. 5.1, we need to convert the single-thread algorithm into parallel algorithms. Then, such a parallel machine learning algorithm is mapped on the computation platform. The advantage of adopting distributed machine learning on IoT devices is further summarized. Firstly, resource constrained IoT devices are fully utilized to perform real-time data analytics such as indoor positioning. The burden for high bandwidth communication between IoT devices and central processors is relieved. Secondly, the leakage of sensitive information is minimized since computation is performed locally. Thirdly, the scalability of IoT system is improved. Distributed computation is scalable to the increasing data size as well as the growing number of IoT devices. As such, in this chapter, we will first discuss the distributed machine learning algorithms

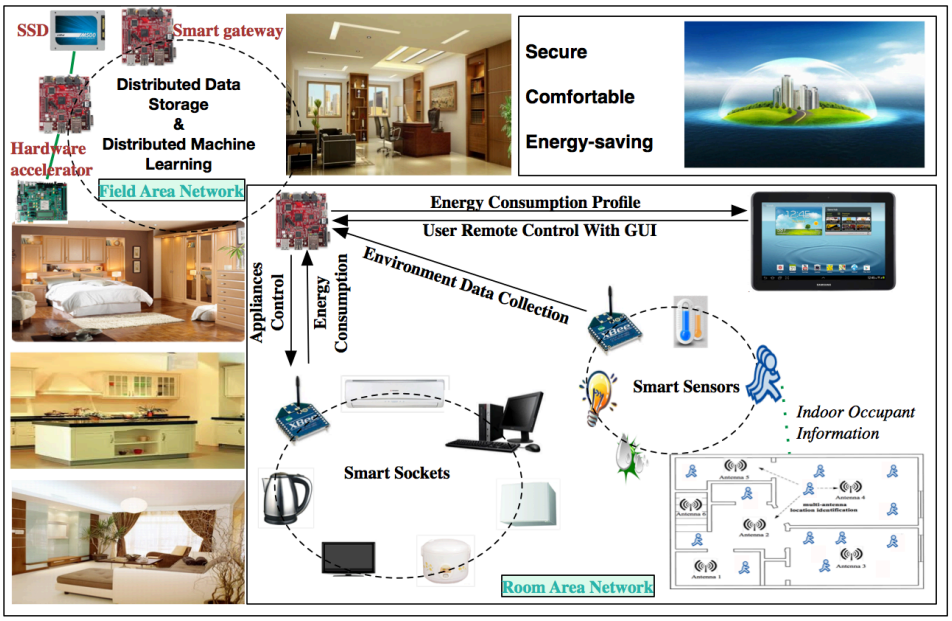


Figure 5.1: The overview of IoT based smart building using smart-gateway network

and then further apply these algorithms on the distributed edge devices for applications such as indoor positioning, energy management and network security.

5.1.1 Indoor Positioning System

Indoor positioning is one of the enabling technologies to help smart building system to anticipate people's needs and assist with possible automate actions [25, 144]. Because of an astonishing growth of wireless systems, wireless information access is now widely available. Wi-Fi-based indoor positioning becomes one of the most popular positioning system up to date due to its low cost and complexity to set up on a gateway network with reasonable accuracy [20, 145].

Many WiFi-data based positioning systems have been developed recently for indoor positioning based on the received signal strength indicator (RSSI) [16]. As the RSSI parameter can show large dynamic change under environmental change (such as obstacles) [16, 21], the traditional machine-learning based WiFi-data analytic algorithms such as K-nearest neighbourhood (KNN), neural network, and support vector machine (SVM) are all centralized with large latency to adapt to the environmental change. This is because the training has high computational complexity [82], which will introduce large latency and also cannot be adopted on the sensor network directly.

As discussed in Chapter 2.2.2, indoor positioning data is critical to analyze occupant behaviors. Therefore, we utilize the distributed machine learning on the WiFi infrastructure in the smart building to build up the indoor positioning system.

5.1.2 Energy Management System

There is an increasing need to develop cyber-physical energy management system (EMS) for modern buildings supplied from the main power grid of external electricity as well as the additional power grid of new renewable solar energy [28]. An accurate short-term load forecasting is crucial to perform load-scheduling and renewable energy allocation based demand-response strategy. However, previous works focus on one or some parts of influential factors (e.g. season factors [31], cooling/heating factors [32] or enviromental factors [33]) without consideration of occupant behaviors. The occupant behavior is one of the most influential factors affecting the energy load demand [1]. Understanding occupant behaviors can provide more accurate load forecasting and better energy saving without sacrificing comfort levels. As such, the building energy management system with consideration of occupant behaviors is greatly required. In this chapter, we will investigate short-term load forecasting with consideration of occupants behavior. Based on this, we further propose energy management system to allocate solar energy to save cost.

5.1.3 Network Intrusion Detection System

Any successful penetration is defined to be an intrusion which aims to compromise the security goals (i.e integrity, confidentiality or availability) of a computing and networking resource [146]. Intrusion detection systems (IDSs) are security systems used to monitor, recognize, and report malicious activities or policy violations in computer systems and networks. They work on the hypothesis that an intruders behavior will be noticeably different from that of a legitimate user and that many unauthorized actions are detectable [147, 148]. Anderson et al. [146] defined the following terms to characterize a system prone to attacks:

- **Threat**: The potential possibility of a deliberate unauthorized attempt to access information, manipulate information or render a system unreliable or unusable.
- **Risk**: Accidental and unpredictable exposure of information, or violation of operations integrity due to malfunction of hardware or incomplete or incorrect software design.
- **Vulnerability**: A known or suspected flaw in the hardware or software design or operation of a systam that exposes the system to penetration of its information to accidental disclosure.

[1] Obviously, the occupant refers to the end users of buildings.

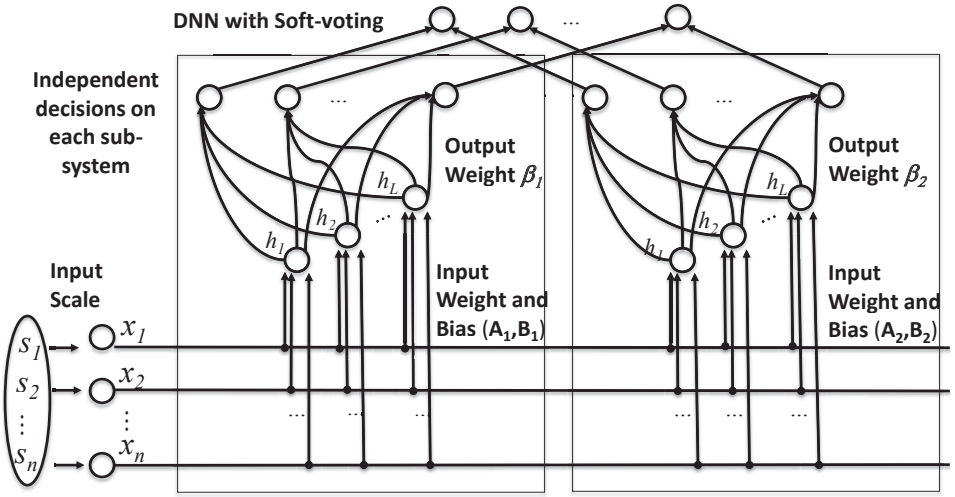


Figure 5.2: Voting based distributed-neural-network with 2 sub-systems

- **Attack**: A specific formulation or execution of a plan to carry out a threat.
- **Penetration**: A successful attack in which the attacker has the ability to obtain unauthorized/undetected access to files and programs or the control state of a computer system.

The aim of cyber physical security techniques such as network intrusion detection system (NIDS) is to provide a reliable communication and operation of the whole system. This is especially necessary for network system such as Home Area Network (HAN), Neighborhood Area Network (NAN) and Wide Area Network (WAN) [149, 150]. Network intrusion detection system (NIDS) is one important mechanism to protect a network from malicious activities in a real-time fashion. Therefore, in IoT networks, a low-power and low-latency intrusion detection accelerator is greatly needed. In this chapter, we will develop a machine learning accelerator on distributed gateway networks to detect network intrusions to provide system security.

In summary, this chapter will propose distributed machine learning algorithms on smart-gateways for IoT based applications, which are indoor positioning system, energy management system and network intrusion detection system.

5.2 Algorithm Optimization

5.2.1 Distributed Neural Network

Our neural network with two sub-systems is shown as Fig. 5.2, which is inspired by extreme learning machine (ELM) and compressed sensing [82, 151]. It is a distributed

neural network (DNN) since each subnetwork is a single layer feed forward neural network (SLFN) and is trained independently with final decisions merged by voting. More details on ensemble learning will be discussed in Chapter 5.2.3. The neural network in the sub-system is trained based on Algorithm 1 in Chapter 3.2. For the completeness purpose, we briefly introduce the training procedure again.

The input weight in our proposed neural network is connected to every hidden node and is randomly generated independent of training data. Therefore, only the output weight is calculated from the training process. Assume there are N arbitrary distinct training samples $\boldsymbol{X} \in \mathbb{R}^{N\times n}$ and labels $\boldsymbol{T} \in \mathbb{R}^{N\times m}$, where $\boldsymbol{X}$ is training data such as scaled RSSI values from each gateway and $\boldsymbol{T}$ is the training label respectively. For our neural network, the relation between the hidden neural node and input training data is

$$\boldsymbol{preH} = \boldsymbol{XA} + \boldsymbol{B},\ \boldsymbol{H} = Sig(\boldsymbol{preH}) = \frac{1}{1+e^{-\boldsymbol{preH}}} \tag{5.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times L}$ and $\boldsymbol{B} \in \mathbb{R}^{N\times L}$. $\boldsymbol{A}$ and $\boldsymbol{B}$ are randomly generated input weight and bias formed by a_{ij} and b_{ij} between $[-1,1]$. $Sig(\boldsymbol{preH})$ refers to the element-wise sigmoid operation of matrix $\boldsymbol{preH}$. $\boldsymbol{H} \in \mathbb{R}^{N\times L}$ is the result from sigmoid function for activation.

In general cases, the number of training data is much larger than the number of hidden neural nodes (i.e. $N > L$). To find the output weight $\boldsymbol{\Gamma}$ is an overdetermined system. Therefore, to estimate the output weight is equivalent to minimizing $||\boldsymbol{T} - \boldsymbol{H\Gamma}||_2$. The general solution can be found as

$$\boldsymbol{\Gamma} = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{T},\ \boldsymbol{H} \in \mathbb{R}^{N\times L} \tag{5.2}$$

where $\boldsymbol{\Gamma} \in \mathbb{R}^{L\times m}$ and m is the number of symbolic classes. $(\boldsymbol{H}^T \times \boldsymbol{H})^{-1}$ exits for full column rank of $\boldsymbol{H}$ [82]. In the inference phase, output node $\boldsymbol{Y}$ is calculated by hidden node output and output weight, given as

$$\boldsymbol{Y} = \boldsymbol{H} \cdot \boldsymbol{\Gamma} \tag{5.3}$$

where $\boldsymbol{\Gamma}$ is obtained by solving incremental least-squares problem. The index of maximum value in $\boldsymbol{Y}$ represents the class that test case belongs to.

Such training method can effectively perform classification and regression task depending on the data type of target $\boldsymbol{T}$. For indoor positioning, we can treat each location area as one class. Then the problem of indoor positioning becomes a classification problem based on the input WiFi RSSI data value. The same logic can also be applied to network detection system by differentiating the normal network traffics from an attack. This is also a classification problem. For the load forecasting problem, it is a regression

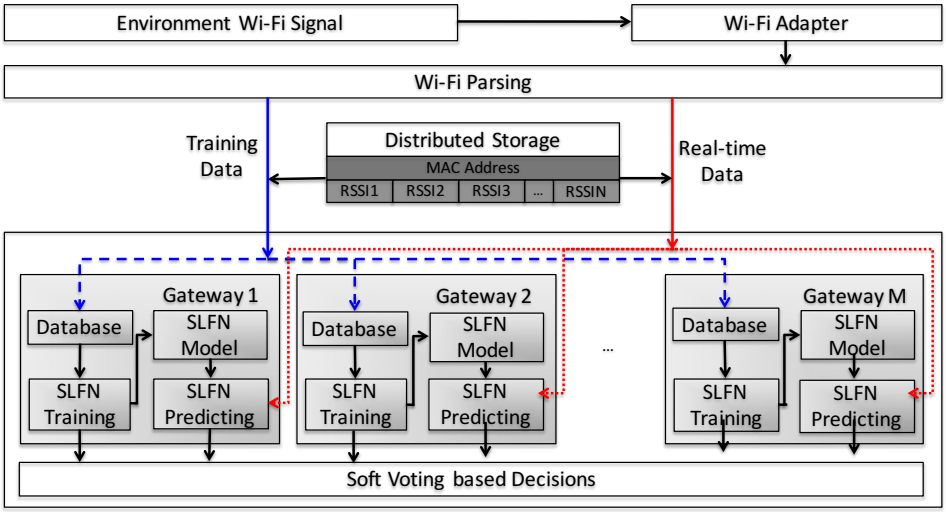


Figure 5.3: Working flow of distributed-neural-network based energy management system

problem since the actual load is a continuous value. The target $\boldsymbol{T}$ should be set as a continuous series for the load prediction. The working flow of distributed-neural-network is summarized in Fig. 5.3.

5.2.2 Online Sequential Model Update

Since new training data such as energy consumptions is time-series data and arrives sequentially as the operation of the system is on, an online sequential update for the machine learning model is necessary. This is especially true for load forecasting since new arrival energy consumption data provides a better indication for future load forecasting than old data. Our proposed online sequential model is updated constantly with new training samples added to the training set $\boldsymbol{X}$. The online sequential model updates include two phases: initialization phase and sequential learning phase [152].

In the initialization phase, the output weight $\boldsymbol{\beta}^{(0)}$ can be calculated based on Algorithm 2, which is

$$\boldsymbol{\beta}^{(0)} = \boldsymbol{S}_0 \boldsymbol{H}(0)^T \boldsymbol{T}_0 \tag{5.4}$$

where $\boldsymbol{S}_0 = (\boldsymbol{H}(0))^T \boldsymbol{H}(0))^{-1}$ can be calculated using the incremental least-squares solver.

In the sequential learning phase, the new training data arrives one-by-one. Given the $(k+1)$-th new training data arrival, the output weight $\boldsymbol{\beta}^{k+1}$ can be calculated as

$$\begin{aligned} \boldsymbol{S}_{k+1} &= \boldsymbol{S}_k - \boldsymbol{S}_k \boldsymbol{H}(k+1)^T (\boldsymbol{I} + \boldsymbol{H}(k+1) \boldsymbol{S}_k \boldsymbol{H}(k+1)^T)^{-1} \boldsymbol{H}(k+1) \boldsymbol{S}_k \\ \boldsymbol{\beta}^{k+1} &= \boldsymbol{\beta}^k + \boldsymbol{S}_{k+1} \boldsymbol{H}(k+1)^T (\boldsymbol{T}_{k+1} - \boldsymbol{H}(k+1) \boldsymbol{\beta}^k) \end{aligned} \tag{5.5}$$

Please note that Cholesky decomposition in Algorithm 2 can also be used to solve matrix inversion problem. Such online sequential learning can enable automatic learning on smart-gateway network for ambient intelligence.

5.2.3 Ensemble Learning

Ensemble learning is to use a group of base-learners to have a higher performance [153]. When combining multiple independent classifiers, the final performance can be enforced. As we have discussed in Chapter 5.2.1, the input weight and bias $\boldsymbol{A}, \boldsymbol{B}$ are randomly generated, which strongly supports that each sub network is an independent expert to make decisions. Each gateway will generate posteriori class probabilities $P_j(c_i|\boldsymbol{x}), i = 1,2,...,m, j = 1,2,...,N_{slfn}$, where $\boldsymbol{x}$ is the received data, m is the number of classes and N_{slfn} is the number of sub-systems for single layer network deployed on smart-gateway. During the inference process, the output of a single layer forward network (SLFN) will be a set of values $y_i, i = 1,2,...,m$. Usually, the maximum y_i is selected to represent its class i. However, in our case, we scale the training and inference input between $[-1,1]$ and target labels are also formed using a set of $[-1,-1,...1...,-1]$, where the only 1 represents its class and the target label has length m. The posteriori probability is estimated as

$$P_j(c_i|\boldsymbol{x}) = (y_i + 1)/2, \ j = 1,2,...,N_{slfn} \tag{5.6}$$

A loosely stated objective is to combine the posteriori of all sub-systems to make more accurate decisions for the incoming data $\boldsymbol{x}$. Under such case, information theory suggests to use a cross entropy (Kullback-Leibler distance) criterion [154], where we may have two possible ways to combine the decisions (Geometric average rule and Arithmetic average rule). The geometric average estimates can be calculated as

$$P(c_i) = \prod_{j=1}^{N_{slfn}} P_j(c_i|\boldsymbol{x}), \ i = 1,2,...m \tag{5.7}$$

and the arithmetic average estimate is shown as

$$\begin{aligned} P(c_i) &= \frac{1}{N_{slfn}} \sum_{j=1}^{N_{slfn}} P_j(c_i|\boldsymbol{x}), \ i = 1,2,...m \\ &= \boldsymbol{W_v}[P_1(c_i|\boldsymbol{x}), P_2(c_i|\boldsymbol{x}), ..., P_{N_{slfn}}(c_i|\boldsymbol{x})]^T \end{aligned} \tag{5.8}$$

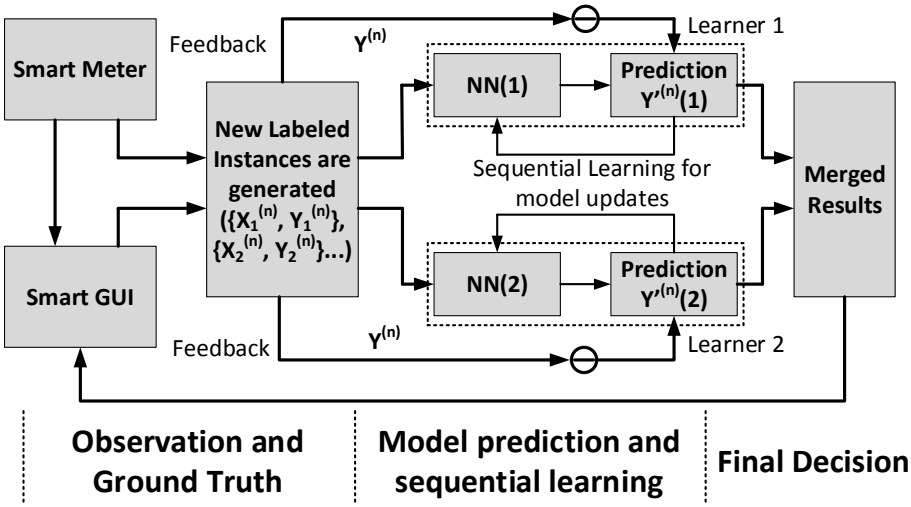


Figure 5.4: Two distributed learners with online sequential learning process

where $P(c_i)$ is the posteriori probability to choose class c_i and we will select the maximum posteriori $P(c_i)$ for both cases. $\boldsymbol{W_v}$ represents the weight of each posteriori probability (soft votes), which is initialized as a vector of $\frac{1}{N_{slfn}}$. In this section, we use arithmetic average as soft-voting of each gateway since [154] indicates that geometric average rule works poorly when the posteriori probability is very low. This may happen when the object to locate is far away from one gateway and its RSSI is small with low accuracy for positioning. We can further update $\boldsymbol{W_v}$ to adjust the weights of each vote during the training phase, where we can assign larger weights for accurate classifiers. The final decision is processed at the central gateway to collect the posteriori probability (soft votes) from each sub-system on other gateways. Such soft-voting will utilize the confidence of each sub-system and can be further adjusted to have weighted voting. This is especially helpful in improving indoor positioning accuracy.

To summarize, the sequence of learning events on the proposed distributed neural network can be described as follows.

1. **Observation:** Each individual learner i observes instances or receives input features.

2. **Local Prediction:** Individual learner generates local prediction $P_k(c_i)$ in a parallel fashion, where k is the index of classifiers.

3. **Final Prediction:** Final prediction is generated based on the weighted voting as $F = \boldsymbol{W_v}[P_1(c_i|\boldsymbol{x}), P_2(c_i|\boldsymbol{x}), ..., P_{N_{slfn}}(c_i|\boldsymbol{x})]^T$.

4. **Feedback:** Each individual learner obtains the true label/profile.

5. **Configuration update:** Perform sequential learning updates based on (5.5).

Fig. 5.4 shows a two learner scheme with online sequential learning process. New labeled data is generated from smart GUI or smart meter. For the indoor positioning application, label is provided by users for training. For the load forecasting application, new load profile received from electric meters will be used as new training data. This ground truth information will be used by each learner.

5.3 IoT based Indoor Positioning System

5.3.1 Problem Formulation

The primary objective is to locate the target as accurate as possible considering the scalability and complexity.

Objective 1: Improve the accuracy of positioning subject to the defined area.

$$\begin{aligned} \min\ & e = \sqrt{(x_e - x_0)^2 + (y_e - y_0)^2} \\ & s.t.\ label(x_e, y_e) \in \boldsymbol{T} \end{aligned} \tag{5.9}$$

where (x_e, y_e) is the system estimated position belongs to the positioning set $\boldsymbol{T}$ and (x_0, y_0) is the real location coordinates. Therefore, a symbolic model based positioning problem can be solved using training set Ω to develop neural network.

$$\Omega = \{(s_i, t_i), i = 1, \cdots, N, s_i \in R^n,\ t_i \in \boldsymbol{T}\} \tag{5.10}$$

where N represents number of datasets and n is the number of smart gateways, which can be viewed as the dimension of the signal strength space. s_i is the vector containing RSSI values collected in the i-th dataset, $t_i \in \{-1, 1\}$ is the label assigned by the algorithm designer. Note that $\boldsymbol{T}$ labels the physical position. The more labels are used, the more accurate the positioning service is.

Objective 2: Reduce the training time on the distributed-neural-network on Hardware BeagleBaord-xM. To distribute training task on n gateways, the average training time should be minimized to reflect the reduced complexity on such gateway system.

$$\begin{aligned} \min\ & \frac{1}{n} \sum_{i=1}^{n} t_{train,i} \\ & s.t.\ e < \varepsilon \end{aligned} \tag{5.11}$$

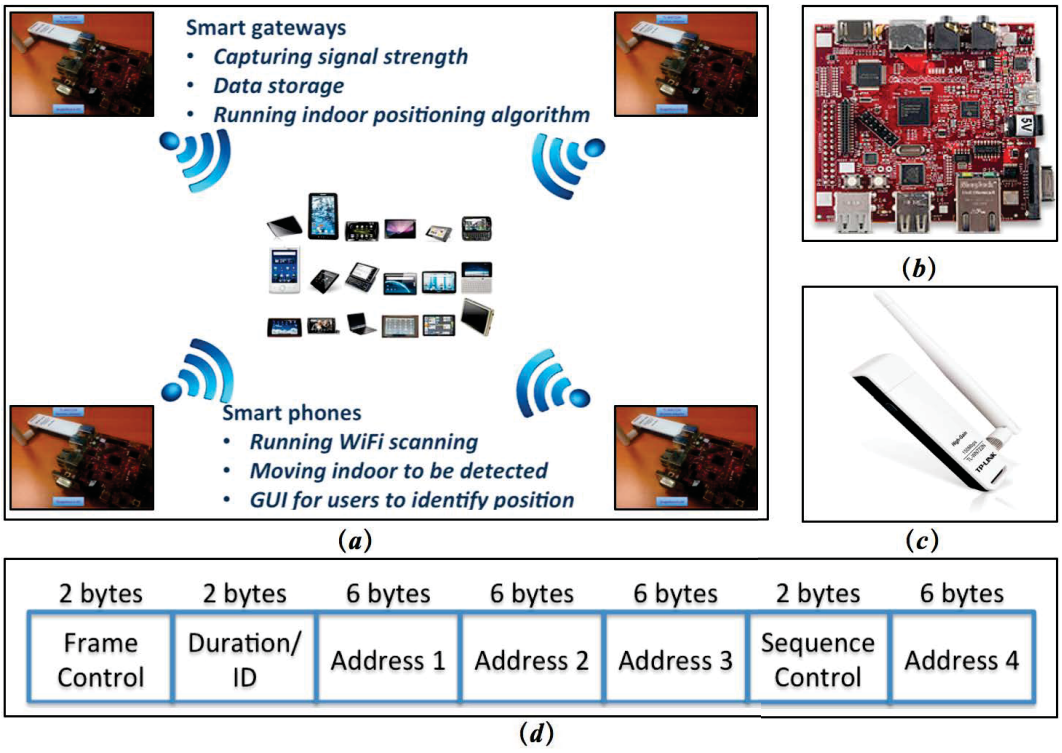


Figure 5.5: (a) Indoor positioning components overview (b) BeagleBoard xM (c) TL-WN722N (d) MAC frame format in WiFi header field

where $t_{train,i}$, refers to the training time of the i-th BeagleBoard. e is the training error and ε is the tolerable maximum error. The symbolic model based positioning is a classification problem, which can be solved by neural networks. To reduce the training time, we use the distributed neural network for better accuracy and computation resource utilization.

5.3.2 Indoor Positioning System

For the physical infrastructure, an indoor positioning system (IPS) by WiFi data consists of at least two hardware components: a transmitter unit and a measuring unit. Here, we use smart-gateways to collect WiFi signal emitted from other smart devices (phone, pad) of moving occupants inside the building. The IPS determines the position by analyzing WiFi-data on the smart-gateway network [155].

The smart-gateway (BeagleBoard-xM) for indoor positioning is shown in Fig. 5.5(b). In Fig. 5.5(c), TL-WN722N wireless adapter is our WiFi sensor for capturing wireless signals. BeagleBoard-xM runs Ubuntu 14.04 LTS with all the processing done on board, including data storage, WiFi packet parsing, and positioning algorithm computation. TL-WN722N works in monitor mode, capturing packets according to IEEE 802.11.

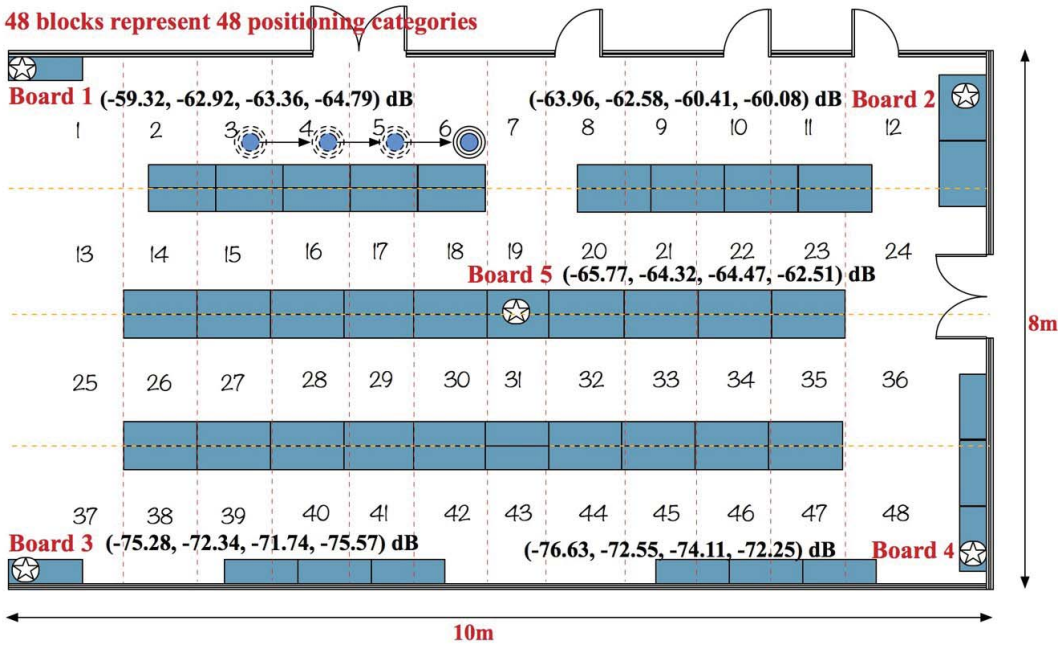


Figure 5.6: An example of position tracking in a floor with 48 blocks representing 48 labels

They are connected with a USB 2.0 port on BeagleBoard-xM.

As depicted in Fig. 5.5(d), WiFi packets contain a header field (30 bytes in length), which contains information about Management and Control Address (MAC). This MAC address is unique to identify the device where the packet came from. Another useful header, which is added to the WiFi packets when capturing frames, is the radio-tap header. This radio-tap header contains information about the RSSI, which reflects the information of distance [145]. With MAC address to identify objects and RSSI values to describe distance information, indoor positioning can be performed.

5.3.3 Experiment results

Experiment Setup Indoor test-bed environment for positioning is presented in Fig. 5.6, with total area being about 80 m^2 (8 m at width and 10 m at length) separated into 48 regular blocks, each block represents a research cubicle. 5 gateways, with 4 at 4 corners of the map, 1 in the center of the map, are set up for the experiment. As shown in Fig. 5.6, 5 gateways will receive different RSSI values as the object moving. To quantify our environment setting, here the positioning accuracy is defined as r, representing radius of target area. It is generated from $S = \pi r^2$, where S is the square of the whole possible positioning area.

Table 5.1: Experimental set-up parameters

Positioning Parameters	Value	Energy Management Parameters	Value
Traing Date Size	18056	**Initial Training Days**	7 Days
Inference Date Size	2000	**House area**	1700 ft^2
Data Dimension	5	**Solar PV area**	25-50 m^2
Number of labels	48	**Continue Evaluation Days**	23 Days
No. of Gateway	5	**Environmental Data**	30 Days
Inference area	80 m^2	**Motion Data**	30 Days

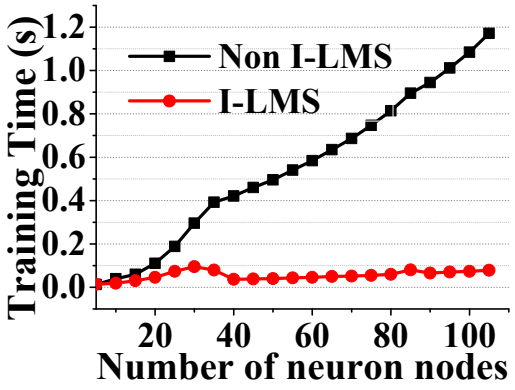


Figure 5.7: Training time for SLFN by Incremental Cholsky decomposition

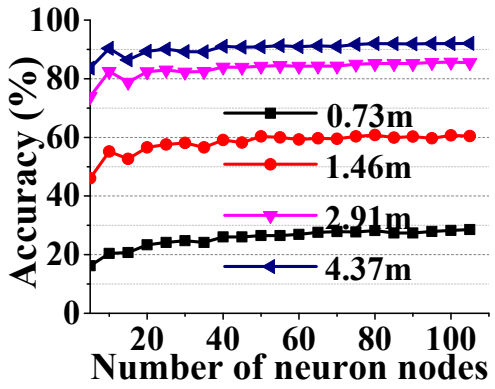


Figure 5.8: Inference Accuracy under different positioning scale

Besides, positioning precision is defined as the probability that the targets are correctly positioned within certain accuracy. The definition is as follows:

$$Precision = \frac{N_{pc}}{N_p} \tag{5.12}$$

where N_{pc} is the number of correct predictions and N_p is the number of total predictions. The summary for the experiment set-up is shown in Table 5.1

Real-time Indoor Positioning Results The result of the trained neural forward network is shown as Fig. 5.7 and Fig. 5.8. The training time can be greatly reduced by using incremental Cholesky decomposition. This is due to the reduction of least square complexity, which is the limitation for the training process. As shown in Fig. 5.7, training time maintains almost constant with increasing number of neural nodes when the previous training results are available. Fig. 5.8 also shows the increasing accuracy under different positioning scales from 0.73m to 4.57m. It also shows that increasing the number of neural nodes will increase the performance to certain accuracy and maintains almost flat at larger number of neural nodes.

Table 5.2: Comparison table with previous works on indoor positioning accuracy

System/Solution	Positioning Algorithms	Precision
Proposed DNN	Neural network	58% within 1.5m, 74%,within 2.2m and 87% within 3.64m
Proposed SV-DNN	Soft-voting and DNN	62.5% within 1.5m, 79%,within 2.2m and 91.2% within 3.64m
RADAR [156]	KNN and Viterbi	50% within 2.5m and 90% within 5.9m
DIT [157]	Neural network and SVM	90% within 5.12m for SVM 90% within 5.40m for MLP
Ekahau [16]	Probabilistic method	50% within 2m (indoors)
SVM [23]	SVM method	63% within 1.5m, 80%,within 2.2m and 92.6% within 3.64m
ANN1 [158]	Neural Network	30% within 1 m and 60% within 1-2m
ANN2 [159]	Neural Network	61% within 1.79 m and 85% within 3m

Table 5.3: Performance precision with variations on proposed DNN with soft-voting

	Test time (s)	Train time (s)	0.73m	1.46m	2.19m	2.91m	3.64m	4.37m	5.1m	No. of Nodes
SVM	9.7580	128.61	31.89%	63.26%	80.25%	88.54%	92.58%	94.15%	94.71%	N.A.
Acc. &Var.			0.530	0.324	0.394	0.598	0.536	0.264	0.0975	
DNN	0.1805	1.065	23.94%	57.78%	74.14%	82.61%	87.22%	90.14%	91.38%	100
Acc. &Var.			1.2153	0.0321	0.1357	0.2849	0.0393	0.0797	0.0530	
SV-DNN (2)	0.1874	2.171	29.36%	61.23%	77.20%	86.24%	90.25%	92.19%	93.14%	2 Sub-systems
Acc. &Var.			0.358	0.937	0.526	0.517	0.173	0.173	0.124	Each 100
SV-DNN (3)	0.1962	3.347	30.52%	62.50%	79.15%	87.88%	91.20%	92.92%	94.08%	3 Sub-systems
Acc. &Var.			0.325	1.952	0.884	1.245	0.730	0.409	0.293	Each 100

Performance Comparison For positioning algorithms, given the same accuracy, the smaller error zone is preferred. Another comparison is that given the same error zone, a better positioning accuracy is desired. In Table 5.2, we can see that although single layer network cannot perform better than SVM, it is able to achieve a state-of-the-art result and outperforms some other positioning algorithms [156, 157, 159]. For example, our proposed algorithm can attain 91.2% accuracy within 3.64m as compared to the 90% accuracy of [157] within 5.4m. Our algorithm also achieves 87.88% accuracy within 2.91m when compared to the 85% accuracy within 3m of [159]. Moreover, by using maximum posteriori probability based soft-voting, SV-DNN can be very close to the accuracy of SVM. Table 5.3 shows the detailed comparisons between proposed DNN positioning algorithm with SVM. Please note that the time reported is the total time for training data size 18056 and inference data size 2000. It shows more than 120x training time improvement and more than 54x inference time saving for proposed SLFN with 1 sub-network comparing to SVM. Even adding soft-voting with 3 sub-networks, 50x and 38x improvement in inference and training time respectively can be achieved. Please note that for fair training and inference time comparison, all the time is recorded using Ubuntu 14.04 LTS system with core 3.2GHz and 8GB RAM. Variances of the accuracy is also achieved by 5 repetitions of experiments and the reported results are the average

values. We find that the stability of proposed DNN is comparable to SVM. Moreover, the inference and training time does not increase significantly with new added subnetworks. Please note that SVM is mainly limited by its training complexity and binary nature where one-against-one strategy is used to maintain the accuracy. This strategy requires to build $m(m-1)/2$ classifiers, where m is the number of classes. Fig. 5.9 shows the error zone of proposed SV-DNN.

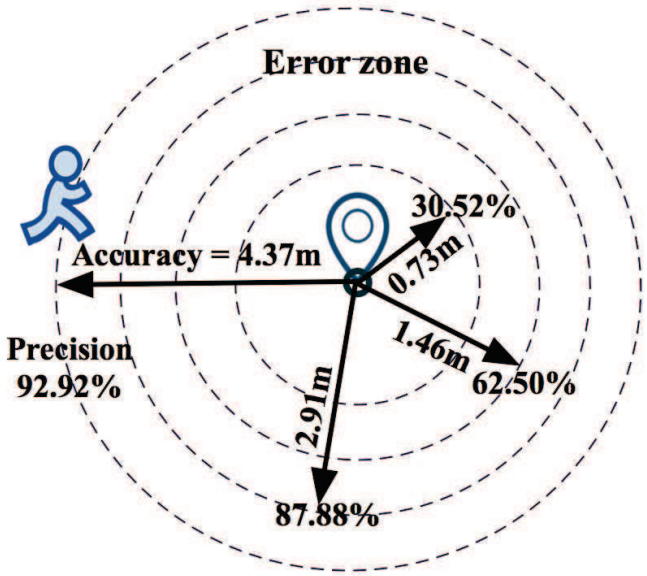


Figure 5.9: Error Zone and accuracy for indoor positioning

5.4 IoT based Energy Management System

During the peak period, peak demand may exceed the maximum supply level that the main electricity power-grid can provide, resulting in power outages and load shedding. Moreover, the electricity price of the main power-grid is higher during the peak period [160, 161]. This is especially true when the pricing strategy is dynamic with peak period price and non-peak period price. Therefore, we schedule the solar energy as the additional power supply to compensate the peak demand in the main electricity power-grid. Using profiles of occupant behaviors and energy consumption, we apply solar energy allocation to rooms with a fast and accurate short-term load prediction.

5.4.1 Problem Formulation

We denote the generated solar energy $G(t)$ and the energy demand $D(t)$ from main electricity power-grid at time t. The allocated solar energy $Al(t)$ is determined by the

energy management system, which is shown as

$$Al(t) = f(\boldsymbol{B}_m, \boldsymbol{E}) \tag{5.13}$$

where the $\boldsymbol{B}_m$ and $\boldsymbol{E}$ are the predicted occupant behavior profile and energy profile respectively.

Here, we formally define the objective of allocating solar energy.

Objective 1: A set of solar energy allocation strategy $Al(t)$ should be determined to minimize the standard deviation of energy consumption from main electricity powergrid with peak load reduction.

$$argmin(dev \sum_{t=1}^{t=24} (D(t) - Al(t))) \tag{5.14}$$

Objective 2: The daily energy cost is expected to be minimized through solar energy allocation strategy. The daily energy cost minimization can be expressed as

$$argmin \sum_{t=1}^{t=24} (D(t) - Al(t))p(t) \tag{5.15}$$

where $p(t)$ represents the electricity price. Please note that the two objectives are aligned when dynamic pricing strategy is set. Dynamic pricing strategy indicates that electricity price is expensive in the peak period and relative cheap in the non-peak period [160, 161]. Therefore, by reducing the peak energy consumption, we actually reduce the electricity usage at peak period to save money. Naturally, a constraint should be taken into consideration that the allocated solar energy $Al(t)$ cannot exceed the available amount of solar energy at anytime.

$$Al(t) \leqslant L(t) \tag{5.16}$$

where $L(t)$ represents the amount of solar energy stored in the energy storage system at time t.

5.4.2 Energy Management System

The overall real-time distributed system for energy management of a smart building is based on hybrid power supply as shown in Fig. 5.10. This EMS infrastructures are implemented inside rooms based on smart-gateway network for collecting and analyzing sensed data [160]. The components of the smart building can be summarized as follows:

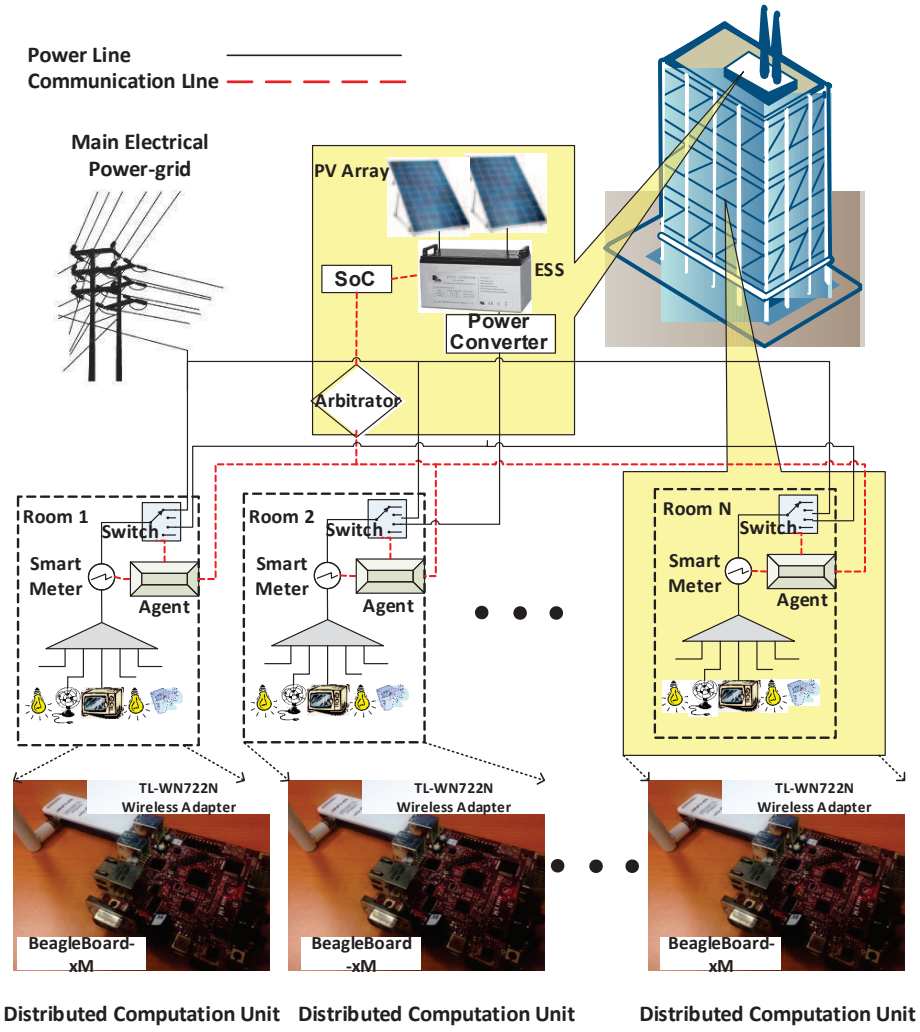


Figure 5.10: Hybrid smart building architecture

- *Main electricity power-grid* is the primary energy supplier for household occupants from external electricity supplier, whose price is much higher than solar energy.

- *Additional electricity power-grid* is the solar photovoltaic panel which is constructed by connected photovaltaic cells [162, 163]. There is also energy storage system used for storing solar energy which can be used in the peak period.

In this distributed system, decision-making is performed independently in each smart gateway. As such, even if one gateway at room level is broken down, the overall system functionality will not be affected. Moreover, by utilizing the solar energy, the EMS can schedule electrical appliances in a room based on the demand-response strategy.

Various sensors are deployed in a smart building system to collect environmental data, energy data and positioning data towards load forecasting and energy management

as shown in Fig. 5.1. Main sensors are listed as follows:

- *Smart power-meter* is referred as energy sensors, which can sense current and record energy consumption data in real time. It can also perform two-way communication with smart device using GUI. Moreover, it includes a switch which is used to change the supplied energy source physically.
- *Wireless adapter* is the WiFi sensor and used to capture the WiFi packets from mobile devices according to IEEE 802.11 protocol. These packets are stored, parsed and computed on BeagleBoard-xM for positioning.
- *Environmental Sensors* include light intensity sensors, temperature sensors and humidity sensors, which are used to monitor the environment and provide data for system decisions.
- *Smart-gateways* are used for storage and computation as the control center. In our system, an open source hardware BeagleBoard-xM with AM37x 1GHz ARM processor is selected for performing intense computation.

Besides above main sensors, the system also includes smart devices with GUI, such as smart phones and tablets, which are used to interact with users and control household appliances. Such GUI can also provide ground true labels to support online sequential model updates.

Real-time Indoor Positioning Environmental WiFi signal is captured by the WiFi Adapter. By parsing the WiFi signal, the data with MAC address and RSSI is stored. Such data is sent to smart-gateway for training with label first. Please note that we divide the whole floor into many blocks and denote each block as a label. The training data and labels are physically collected using our developed GUI on the Android pad. A distributed neural network with multiple single layer feed forward network (SLFN) is trained based on the WiFi data, which is elaborated in Chapter 5.2.1. A small data storage is required to store trained weights for the network. In the real-time application, the same format of Wi-Fi data will be collected and sent into the well trained model to determine its label, which indicates its position. The predicted positioning result will be stored and used for occupant behavior analysis.

Occupant Behavior Profile As shown in Fig. 2.5, occupant position is one of the major driving forces for energy related behaviors. Occupant position and active occupant motion indicate the high level of energy related behaviors in the room [164]. Rooms

inside the same house have vastly different occupant behavior profiles due to different functionalities. Therefore, we extract behavior profiles for different rooms respectively. For each room i, there are four states represented by S for occupant positioning:

$$S = \begin{cases} s_1 : 0 & \text{no occupant in the room } i \\ s_2 : 0 \longrightarrow 1 & \text{occupants entering the room } i \\ s_3 : 1 & \text{occupants in the room } i \\ s_4 : 1 \longrightarrow 0 & \text{occupants leaving the room } i \end{cases} \tag{5.17}$$

where motion state S is detected by indoor positioning system via WiFi data every minute [2]. The probability of occupant motion for room i can be expressed as

$$b_m(t) = \frac{Ti(s_2) + Ti(s_3)}{Ti}, t = 1,2,3,...,96 \tag{5.18}$$

where $Ti(s_j)$ represents the time duration with corresponding state s_j. $b_m(t)$ is occupant motion probability of room i in Ti time interval. Here, we set the Ti as 15 minutes resulting in 96 intervals in one day. Based on the motion probability $b_m(t)$, the probability of active behavior can be expressed as

$$\boldsymbol{B}_m = [b_m(1), b_m(2), b_m(3), ..., b_m(96)] \tag{5.19}$$

Since the occupant behavior profile $\boldsymbol{B}_m$ on the d-th day is extracted based on trend of previous days, we choose previous seven-day behavior profiles as training data $\boldsymbol{X_B}$ and the profile on the 8-th as $\boldsymbol{Y_B}$, which can be shown as

$$\boldsymbol{X_B} = \{\gamma_d \sigma \cdot \boldsymbol{T}(d) | 1 \leqslant d \leqslant 7\} \qquad \boldsymbol{Y_B} = \{\boldsymbol{B}_m(d) | d = 8\} \tag{5.20}$$

where, $\boldsymbol{B}_m(d)$ represents the occupant behavior profile on the d-th day and $\boldsymbol{T}(d) = \boldsymbol{B}_m(d)^T$. Here, γ_d is the daily weight assigned to each training data and σ is the weaken factor used to reduce the effect of abnormal data. Since the latest actual sample has most significant effect on prediction, the arriving new data should be paid more attention using heavy weight. The daily weight γ is set as

$$\gamma_{d-1} > \gamma_{d-2} > ... > \gamma_{d-7} \tag{5.21}$$

Moreover, it is possible that some abnormal data samples appear suddenly. According to historical occupant behavior profiles, extremely abnormal samples can be detected

[2] State s_2 and s_4 are designed mainly for automatic control such as lighting. They can be determined based on the tracking result.

Table 5.4: Input features for short-term load forecasting

Inputs	Descriptions
1	Datet ype: weekday is represented by 1 and weekend is represented by 0
2-25	$Ti(d-7,t)$, $Ti(d-6,t)$, $Ti(d-5,t)$, $Ti(d-4,t)$, $Ti(d-3,t)$, $Ti(d-2,t)$, $Ti(d-1,t)$: Temperature of the seven days preceding to the forecasted day at the same hour
26-49	$H(d-7,t)$, $H(d-6,t)$, $H(d-5,t)$, $H(d-4,t)$, $H(d-3,t)$, $H(d-2,t)$, $H(d-1,t)$: Humidity of the seven days preceding to the forecasted day at the same hour
50-73	$Ld(d-7,t)$, $Ld(d-6,t)$, $Ld(d-5,t)$, $Ld(d-4,t)$, $Ld(d-3,t)$, $Ld(d-2,t)$, $Ld(d-1,t)$: Energy consumption of the seven days preceding to the forecasted day at the same hour

using the average motion probability, which is shown as

$$\sigma = \begin{cases} 1 & \frac{1}{2}Bd \leqslant \bar{\boldsymbol{B}}_m \leqslant \frac{3}{2}Bd \\ 0.1 & \bar{\boldsymbol{B}}_m < \frac{1}{2}Bd \text{ or } \bar{\boldsymbol{B}}_m > \frac{3}{2}Bd \end{cases} \quad s.t \;\; \bar{\boldsymbol{B}}_m = \frac{\sum_{t=1}^{96} b(t)}{96} \quad Bd = \frac{\sum_{d=1}^{7} \bar{\boldsymbol{B}}_m(d)}{7} \tag{5.22}$$

where $\frac{3}{2}Bd$ is defined as the upper bound and $\frac{1}{2}Bd$ is defined as the lower bound.

Energy Profile Feature Extraction

Energy profile $\boldsymbol{E}$ consists of the hourly energy consumption $e(t)$ in the time span of 24 hours. The characteristic of energy profile $\boldsymbol{E}$ in different weather conditions and different day types varies significantly . The energy profile $\boldsymbol{E}$ can be expressed as

$$\boldsymbol{E} = [e(1), e(2), ..., e(24)] \tag{5.23}$$

To forecast the energy profile on the $(d+1)$-th day, we choose training data $\boldsymbol{X_E}$ with features as shown in Table 5.4 and $\boldsymbol{Y_E}$ as the actual energy profile on the d-th day. This is exactly time series data, where new data is obtained sequentially from the utility. Therefore, we can perform sequential model update to capture the change of energy profiles.

Energy Management by Fusing Occupant Behavior and Energy Profile

To formally illustrate the proposed energy management method, two factors should be described firstly. $P_B(t)$ represents the probability of active occupant motion for the whole

house at the time t:

$$P_B(t) = \sum_{i=1}^{M} \alpha_i \theta_i \boldsymbol{B}_m^i(t), \;\; 0 \leqslant \alpha_i \leqslant 1 \tag{5.24}$$

where $\boldsymbol{B}_m^i(t)$ is the occupant motion probability for room i and θ_i is the statistical information of occupants staying in room i indicating the probability of energy related behaviors. M is the number of rooms. Since household appliances in each room are diverse resulting in various energy consumption levels, weight parameter α_i is set according to the energy consumption characteristic of each kind of room. $P_E(t)$ is the proportion of energy consumption at time t:

$$P_E(t) = \frac{e(t)}{\sum_{t=1}^{24} e(t)} \tag{5.25}$$

where $e(t)$ is the component of the predicted 24-hour energy profile. Since the energy consumption is likely to increase when occupant motion probability is increasing [164], the peak load period can be detected by fusing $P_B(t)$ and $P_E(t)$. The probability of load peak $P_{peak}(t)$ can be denoted as

$$P_{peak}(t) = \eta P_B(t) + (1-\eta) P_E(t), 0 \leqslant \eta \leqslant 1 \tag{5.26}$$

When $P_{peak}(t)$ exceeds more than two thirds of the highest value, it is regarded as the load peak moment. The threshold can be determined by end users. At such moment, solar energy will be allocated to alleviate the load from main electricity power-grid. The expected amount of solar energy allocation $A_E(t)$ can be determined by

$$A_E(t) = \Phi(\boldsymbol{E(t)} - \bar{\boldsymbol{E}}) + \Psi P_B(t) \tag{5.27}$$

where the parameter Φ and Ψ are set by occupants according to actual energy demand. $\bar{\boldsymbol{E}}$ is the average energy consumption during 24 hours. The actual amount of solar energy allocation is

$$Al(t) = \begin{cases} A_E(t) & A_E(t) \leqslant (G(t) + L(t-1)) \\ L(t-1) + G(t) & A_E(t) > (G(t) + L(t-1)) \end{cases} \tag{5.28}$$

Here, $L(t-1)$ is the total amount of remaining solar energy until time $t-1$. $G(t)$ is the generated solar energy. $G(t) + L(t-1)$ represents the available amount of solar energy at the time t. If the generated solar energy $G(t)$ is more than the allocated solar energy

$A(t)$, the remaining solar energy $L(t)$ will be accumulated for the next allocation period:

$$L(t) = L(t-1) + (G(t) - Al(t)) \tag{5.29}$$

Fig. 5.11 summarizes the system flow based on the proposed machine learning on distributed smart-gateway platform.

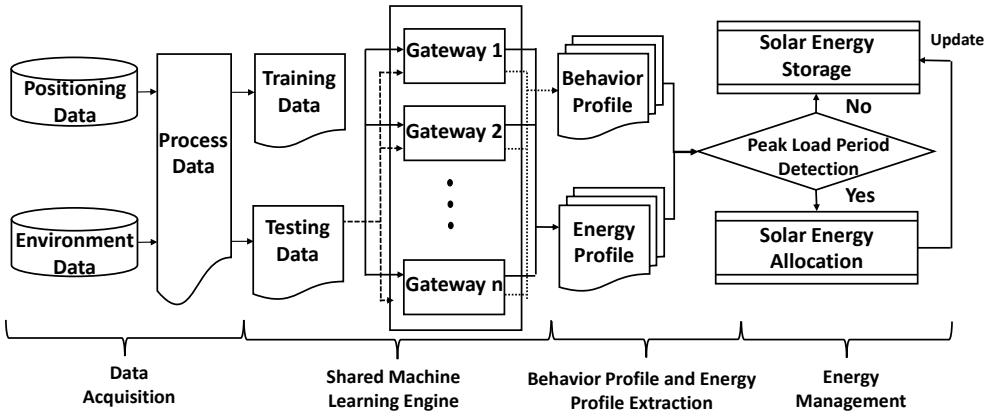


Figure 5.11: Flow based on distributed machine learning on smart-gateway networks

5.4.3 Experiment Results

Experiment Set-up In this experiment, we evaluate our proposed online sequential machine learning engine on distributed smart-gateway networks. We firstly perform occupant profile feature extraction by real-time indoor positioning. Based on the energy profile features, we evaluate the short-term load forecasting in comparison with SVM [165]. Then we verify the proposed energy allocation method in comparison with SVM based predictions. We choose SVM as the baseline for three reasons. Firstly, IoT devices are relatively resource-constrained with limited memory. Complicated neural network such as deep neural network may not be able to map on IoT devices. SVM is relatively light-weight and performs reasonably well. Secondly, the mathematical model of SVM is very clear. SVM fits a hyperplane between 2 different classes given a maximum margin parameter. This algorithm is well studied and well-known to many. Lastly, many works had proposed using SVM for short-term load forecasting [165] , indoor positioning [23] and network intrusion detection [166]. This provides us a good stage to showcase the performance improvement we had achieved over the existing works.

We utilize one dataset from the Smart* Home Dataset [167] and one dataset provided by Energy Research Institute of Nanyang Technological University [3]. The first 7 days

[3]This dataset consists of 24-hour energy consumption data and environmental factor records from

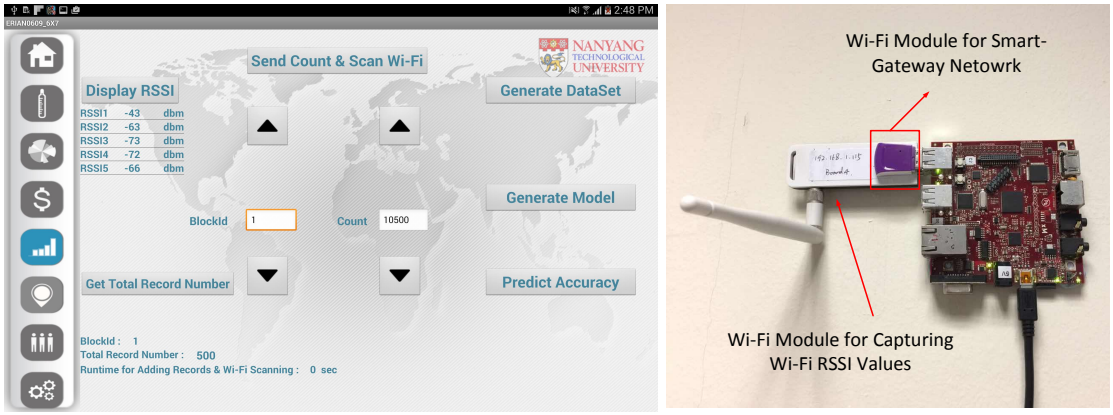


Figure 5.12: Smart-gateway network set-up with GUI for training

are used for initial training with actual energy consumption as ground truth references and the next 23 days are used for inference. The input data for training is summarized in Table 5.4. To illustrate the capability of the proposed method for the energy cost saving, we choose the dynamic electricity pricing strategy from [161] and the solar energy generation profile from [168].

For the occupant behavior profile, it is based on indoor positioning system mentioned in Chapter 5.4.2. To evaluate the indoor positioning system, we have performed an additional experiment to verify its precision. The indoor test-bed environment for positioning is presented in Fig. 5.6, with total area being about 80 m^2 ($8m$ at width and $10m$ at length) separated into 48 regular blocks, each block represents a research cubicle. 5 gateways, with 4 at 4 corners of the map, 1 in the center of the map, are set up for experiment. As shown in Fig. 5.6, 5 gateways will receive different RSSI values as the object is moving. The definition for positioning precision is as follows

$$Precision = \frac{N_{pc}}{N_p} \tag{5.30}$$

where N_{pc} is the number of correct predictions and N_p is the number of total predictions. Fig. 5.12 shows the Android pad based GUI and one single gateway with WiFi module. This GUI can be used to denote labels as well as collect WiFi data. The smart gateway is attached to the wall at four corners and one in central cubical. We use the same experiment set-up in Chapter 5.3.3 as shown in Table 5.1.

For the energy consumption profile, we first predict 24-hour occupant behavior profile using previous 7-day motion probability based on the indoor positioning system and then forecast 24-hour energy consumption using environmental factors. We estimate the prediction accuracy of load forecasting using mean absolute percentage error (MAPE)

2011 to 2015.

and root mean square error (RMSE). MAPE and RMSE are defined as

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{P(t)-R(t)}{R(t)}\right|, \quad RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(P(t)-R(t))^2} \tag{5.31}$$

where $P(t)$ is the prediction value and $R(t)$ is the actual value. Based on the predicted behavior profile and energy profile, we allocate solar energy for reducing peak load demand and saving energy cost.

Occupant Profile Feature Extraction As we have discussed in Chapter 5.4.2, occupant behavior profile is analyzed based on the real-time indoor positioning. After collecting occupant positioning data, we can use this data to build the occupant behavior profile. we predict 24-hour occupant behavior profiles in six rooms: basement, bedroom, guest room, kitchen, living room and master room. Fig. 5.13 shows the predicted motion probability in 15 minutes interval of living room. Corresponding training data and measured data are also displayed to demonstrate the prediction accuracy. We can observe that the predicted motion of occupant is the same trend as the actual occupant motion. By combining profiles of six rooms, the daily behavior profile for the whole house can be estimated.

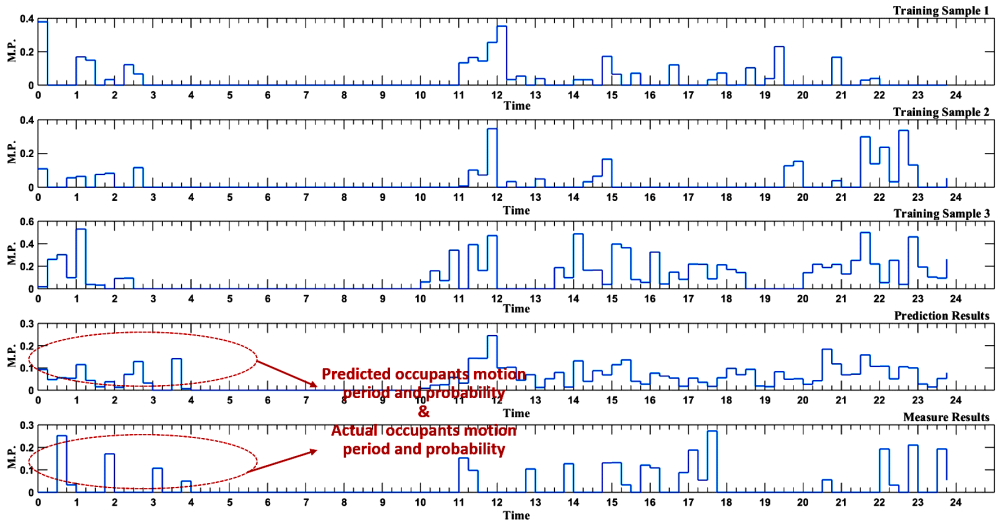


Figure 5.13: Predicted Motion probability within 15 minutes interval in a living room

Energy Profile Feature Extraction Fig. 5.14 illustrates the 24-hour load forecasting results using our proposed method and SVM respectively. It clearly indicates the

Table 5.5: Prediction Accuracy Comparison with SVM

Machine Learning	MAPE			RMSE		
	Max	Min	Avg	Max	Min	Avg
DNN	0.23	0.10	0.15	29.37	10.92	20.30
SVM	0.34	0.14	0.20	33.75	15.72	23.14
Imp.(%)	31.20	0.50	14.83	30.53	3.32	14.60

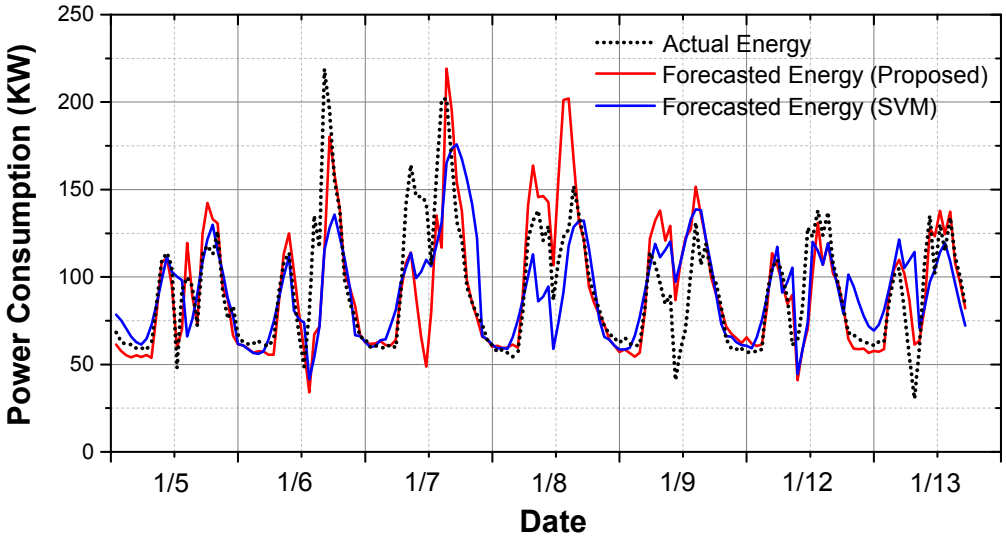


Figure 5.14: Short-term load forecasting with comparison of SVM

energy demand between different days varies but our proposed method can accurately capture the peak time of the energy demand. To verify the forecasting accuracy, the proposed DNN is compared with SVM using MAPE and RMSE, which are shown in Table 5.5. Our proposed DNN forecasting method achieves 14.83% and 14.60% average improvement comparing to SVM in both MAPE and RMSE. Based on such energy demand prediction, we can effectively utilize the solar energy allocation to reduce the peak demand.

Energy and Peak Reduction As mentioned above, it is crucial to reduce the peak energy demand to save cost and maximize the benefit of solar energy from the user perspective. Fig. 5.15 shows the peak reduction based on our proposed method and static energy allocation method respectively. It indicates that our method achieves a lower standard deviation of energy profile from main electricity power-grid than that of static method with more flat curve. The reduced peak loads and cost saving with various solar PV areas are shown in Table 5.6 and Table 5.7 in comparison with SVM based load forecasting. It shows that our method can outperform SVM with peak load reduction from 1.84% to 19.66% and energy cost saving from 2.86% to 26.41% under

various solar areas. Please note the electricity price is dynamic with peak period (8:00-12:00 and 14:00-21:00) 22 USD cents and non-peak period 8 USD cents [161]. The cost saving objective aligns with peak load reduction.

Table 5.6: Energy peak reduction comparisons for DNN and SVM over one month

	SVM			**DNN**			
area (m^2)	Max	Min	Avg	Max	Min	Avg	Imp. (Avg)
25	1943.7	1792.76	1905.97	1943.7	1670.62	1871.48	1.84%
30	2332.44	2130.75	2264.45	2332.44	1811.98	2150.62	5.29%
35	2721.18	2272.11	2601.99	2721.18	1953.34	2353.83	10.54%
40	3109.92	2413.47	2888.84	2908.96	2094.7	2508.93	15.14%
45	3498.66	2554.83	3137.39	3050.32	2236.06	2650.29	18.38%
50	3887.4	2696.19	3340.60	3191.68	2377.42	2791.65	19.66%

Table 5.7: Energy cost saving comparisons for DNN and SVM over one month

	SVM			**DNN**			
area (m^2)	Max	Min	Avg	Max	Min	Avg	Imp. (Avg))
25	88.35	70.19	81.48	88.35	58.43	79.21	2.86%
30	106.02	87.86	98.78	106.02	67.27	95.36	3.59%
35	123.69	104.40	115.21	123.69	76.10	105.11	9.61%
40	141.36	113.24	130.03	127.01	84.94	110.54	17.63%
45	159.03	122.07	142.05	132.03	93.77	114.96	23.57%
50	176.7	130.91	150.91	140.87	96.70	119.38	26.41%

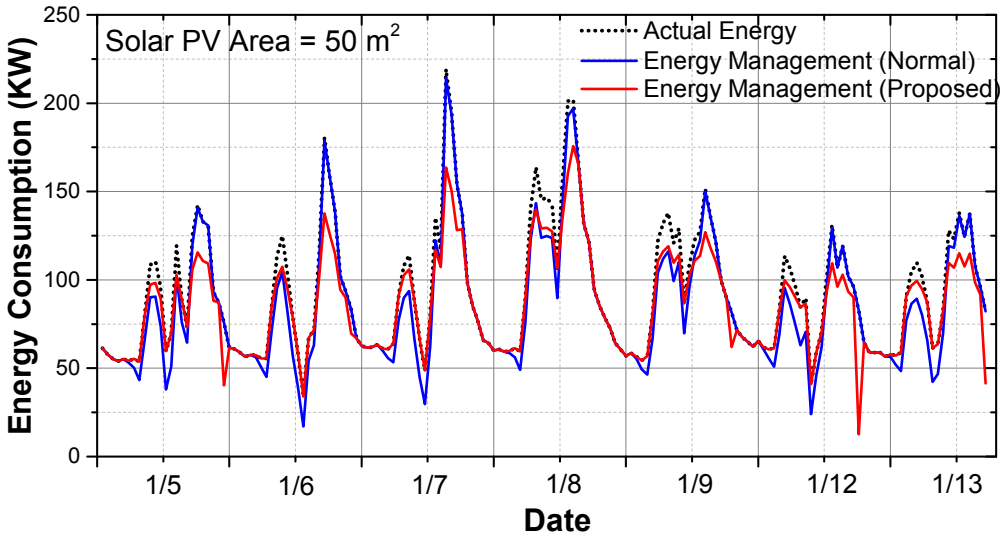


Figure 5.15: Demand response with peak demand reduction

5.5 IoT based Network Security System

5.5.1 Problem Formulation

The goal of network intrusion detection system is to differentiate anomalous network activity from normal network traffic in a timely fashion. The major challenge is that the patterns of attack signatures change over time and the NIDS has to upgrade to handle these changes [38, 169, 170]. Therefore, we design our NIDS with three objectives: high-accuracy, fast-detection and good-adaptivity.

Objective 1: Increase overall anomaly intrusion detection accuracy F-measure defined as:

$$\begin{aligned} F{-}measure &= 2 * \frac{precision * recall}{precision + recall} \times 100 \\ where\ precision &= \frac{tp}{tp+fp}\ ,\ recall = \frac{tp}{tp+fn} \end{aligned} \tag{5.32}$$

where tp, fp and fn represent true positive rate, false positive rate and false negative rate respectively. Following terms are defined to mathematically describe this objective.

1. *False Positives (fp):* Number of normal instances which are detected as intrusions.
2. *False Negatives (fn):* Number of intrusion instances which are detected as normal.
3. *True Positives (tp):* Number of correctly detected intrusion instances.

Since precision and recall are often inversely proportional to each other, F-measure is a better metric of accuracy since it represents the harmonic mean of precision and recall.

Objective 2: Reduce intrusion detection latency W_{Total} at HAN layer for timely system protection.

$$W_{Total} = \frac{1}{\mu - \alpha} \tag{5.33}$$

where α and μ are average packet arrival rate and NIDS system service rate respectively.

Objective 3: Improve the adaptivity of NIDS by developing online sequential learning algorithms. We tackle the objective 1 and 2 by developing a single hidden layer neural network on FPGA. Furthermore, the least-squares solver can be re-used for sequential learning to improve the adaptivity of NIDS. The sequential learning process is discussed in Chapter 5.2.2.

5.5.2 Network Intrusion Detection System

IoT devices such as smart sensors, home appliances are linked to establish home area network (HAN) at the customer layer. Inside HAN, Zigbee (802.15.4) and Wi-Fi are

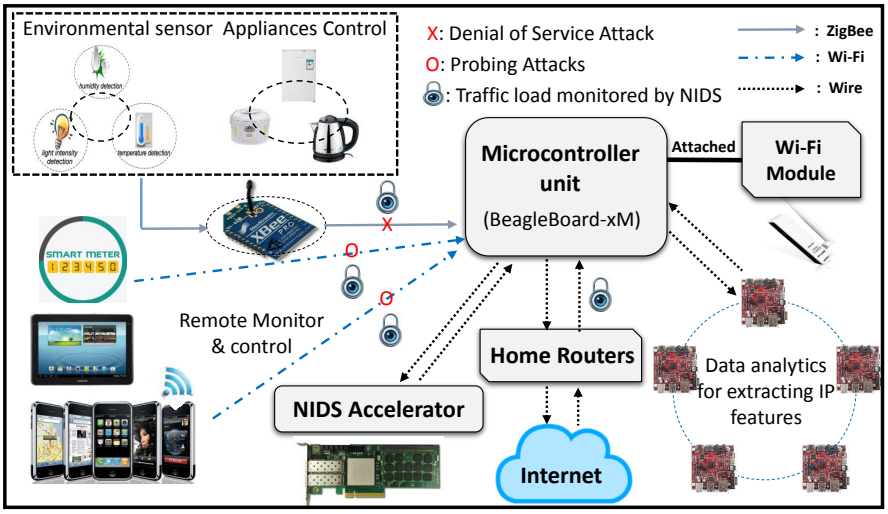


Figure 5.16: IoT network intrusion detection system architecture under cyber attacks

widely adopted communication standards [150] to facilitate remote control and monitoring. Figure 5.16 describes IoT based smart home at the HAN layer. As such, our IoT system consists of the following components:

1. *Smart gateways:* Distributed smart gateways form the control centers of smart buildings and homes to store and analyze data. One can build a smart-gateway based on Beagleboard-Xm [98] with an ARM processor (clocked at 1 GHz and 512 MB RAM) or a Xilinx FPGA (such as Zynq-7000). These smart gateways are also equipped with Zigbee and Wi-Fi communication modules to communicate with smart sensors for information such as light intensity, temperature and humidity data.

2. *Smart sensors:* Smart sensors collect current information from buildings, home appliances and environments. They are able to be remotely controlled by smart gateways.

3. *HAN intrusion detection system (IDS) module:* The IDS module at the HAN level will track the network traffic for intrusions [150]. The distributed smart gateway can perform packet sniffer and extract IP features. The FPGA accelerator on the smart gateway performs fast active monitoring of network traffic for IDS.

As shown in Fig. 5.16, our NIDS architecture will perform intrusion detection from IoT devices to the control center (distributed gateway networks) as well as routers to control center. All packets from external devices will be first sniffed, followed by performing feature extraction and then sent to FPGA accelerator to detect intrusions. As

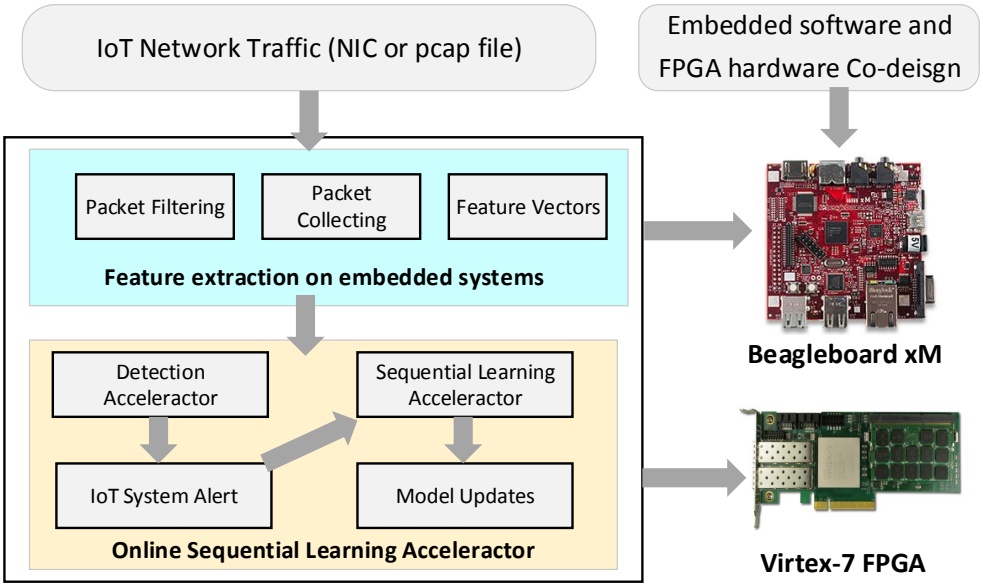


Figure 5.17: Hardware accelerator design on embedded system for IoT network intrusion detection

such active monitoring, a fast intrusion detection process is critical for high performance IoT system with timely reaction to intrusions.

Fig. 5.17 shows the hardware accelerator design on embedded system for IoT NIDS. Connecting PCIe to SoC requires a costumed interface through Multi-Media-Card (MMC). We can also use Xilinx Zynq-7000 Series board featuring ARM core and programmable logics. The targeted devices can be selected based on the cost and specifications. We perform hardware/software partition as feature extraction on software and online-sequential learning on hardware. This is mainly due to the various communications protocols in IoT systems and high complexity of machine learning. Feature extraction using software based embedded system can support more communication protocols and prepare the correct data format for hardware accelerator. Furthermore, such partition increases the flexibility to integrate new threats and the FPGA based machine learning can perform fast sequential learning to build a new model to support the growing complexity of IoT systems.

Resource-constrained IoT devices are connected through wireless communication for remote control and monitoring. However, these communication technologies can introduce new vulnerabilities and security issues into the smart home even when these IoT devices are protected by authentication and encryption. For example, Zigbee technology is vulernable to DOS attacks [171]. Smart meters are also vulnerable to wireless probing and thus consumer metering data can be compromised. Attackers can easily login to modify measurements and control command information which can cause a significant

error in power measurements and a lack of power supply [171]. In HAN, an infiltration of the network from the inside of the network is also possible using a combination of probing attacks, buffer overflow attacks and SQL injection attacks.

Hence it is important to devise a cybersecurity strategy, which can protect against these intrusions. As shown in Fig. 5.16, we will discuss two major attacks in home area network (HAN).

DoS attacks in IoT Network : Denial-of-service (DoS) attack is designed to attack the network by flooding it with useless traffic. IoT devices such as smart meters are vulnerable to DoS during the wireless communication [150]. In a similar fashion, a distributed DOS (DDoS) attack using a botnet can also be launched against IoT devices. A repeatedly jammed communication channel of IoT devices may get them into an endless loop for delivering its data resulting in battery exhaustion or greatly reduced battery life [172].

Probe attacks in IoT Network : Probe attacks are attempts to gain access and acquire information of the target network from an external source. Such attack may take advantage of flaws in the firmware to gather sensitive information [173]. Attackers may also login to IoT devices to modify measurement or control command information leading to system failure.

5.5.3 Experiment Results

In this section, we first discuss the experiment setup and benchmarks following by the machine learning accelerator architecture and resource usage. Then network intrusion detection accuracy and delay analysis are presented. Finally, the energy consumption and speed-up of proposed accelerator are evaluated in comparison with CPU and embedded system.

Experiment Setup and Benchmark To verify our proposed architecture, we have implemented it on Xilinx Virtex 7 [89]. The HDL code is synthesized using Synplify and the maximum operating frequency of the system is 54.1 MHz under 128 parallel PEs. The critical path is identified as the floating-point division, where 9 stages of pipeline are inserted for speedup. Experiment results on our accelerator are denoted as hardware-accelerated NIDS. We develop two baselines to compare the performance.

Baseline 1: General processor (x86 CPU). The general CPU implementation is based on C program with an Intel Core -i5 *3.20GHz* core and *8.0GB* RAM computer. Experiment results on this platform are denoted as software-NIDS.

Table 5.8: NSL-KDD and ISCX-2012 benchmark set-up parameters

Parameter	NSL-KDD [175]	ISCX [174]
Training Data Size	74258	103285
Inference Data Size	74259	103229
Data Dimension	41	11
Number of Labels	Binary (2)	Binary (2)
	MultiClass (4)	MultiClass (5)

Table 5.9: NSL-KDD Data Preprocessing

Feature	**Numeric-valued Transformation**
protocol type	tcp = 1, udp = 2, icmp = 3
service	aol=1, auth=2, bgp=3, courier=4 csnet_ns=5, ctf=6, $\cdots$, Z39_50 = 70
flag	OTH=1, REJ=2, RSTO=3, RSTOS0=4 RSTR=5, S0=6, S1=7, S2=8, S3=9, SF=10, SH=11
IDS(2)	normal=1, attack=2,
IDS(5)	normal=1, probe=2, dos=3, u2r=4, r2l=5

Baseline 2: Embedded processor (ARM CPU). The embedded CPU (Beagle-Board-xM) [98] is equipped with 1GHz ARM core and 512MB RAM. The implementation is performed using C program under Ubuntu 14.04 system. Experiment results on this platform are denoted as embedded-NIDS.

To test our proposed system, we have used two benchmarks ISCX-2012 [174] and NSL-KDD [175] dataset, which include all the attacks mentioned such as DoS, Probe and R2L. Details of each benchmarks are shown in Table 5.8. The data pre-process for network traffic is summarized in Table 5.9 as the input of neural network. We did not consider U2R attack in the NSL-KDD dataset since it will not happen in the home area network (HAN) and the number of samples is very small.

Scalable and Parameterized Accelerator Architecture The proposed accelerator architecture features great scalability for different available resources. Table 5.10 shows all the user-defined parameters supported in our architecture. At circuit level, users can adjust the stage of pipeline of each arithmetic to satisfy the speed, area and resource requirements. At architecture level, the parallelism of PE can be specified based on the hardware resource and speed requirement. The neural network parameters n, N, H can also be reconfigured for specific applications.

The resource utilization under different parallelism is observed from Xilinx ISE after place and routing. From Table 5.11, we can observe that LUT and DSP are almost linearly increasing with parallelism. However, Block RAM keeps constant with increasing parallelism. This is because Block RAM is used for data buffer, which is determined

Table 5.10: Tunable parameters of the hardware accelerator

Parameters		Descriptions
Circuits	{MAN EXP}	Word-length of mantissa, exponent
	$\{P_A, P_M, P_D, P_C\}$	Pipe. stages of adder, mult, , div and comp
Architectures	P	Parallelism of PE in VC
	n	Maximum signal dimensions
	N	Maximum training/inference data size
	H	Maximum number of hidden nodes

Table 5.11: Resource utilization under different parallelism level ($N = 1024$, $H = 1024$, $n = 64$ and $50Mhz$ clock)

Paral.	LUT	Block RAM	DSP
8	65791 (15%)	1311 (89.2%)	43 (1.19%)
16	77042 (17 %)	1311 (89.2%)	59 (1.64%)
32	100571 (23 %)	1311 (89.2%)	89 (2.47%)
64	153108 (35 %)	1311 (89.2%)	152 (4.22%)
128	245292 (56 %)	1311 (89.2%)	280 (7.78%)

by other architecture parameters (N,H,n). For large dataset, external data memory is required.

NIDS Accuracy Analysis Fig. 5.18 shows the classification accuracy with increasing number of hidden nodes. Please note that the binary class is to detect normal and anomaly classes. Clearly, the more hidden nodes, the better accuracy will be. But it saturates at around 450 hidden nodes. Table 5.12 shows the detailed accuracy of each class accuracy. Our proposed neural networks work better than SVM based method and achieve 75.15 % multi-class detection accuracy in the NSL-KDD dataset.

Fig. 5.19 shows the F-measure accuracy (defined in (5.32)) on benchmarks ISCX-2012 and NSL-KDD. We can find that the hardware-accelerated NIDS is slightly less accurate comparing to software-NIDS. This is mainly due to the 8-bit fixed data format. We choose 8-bit width fixed point data format for input to save memory size of the BRAM and DRAM. Fig. 5.19 also shows that our system can not only detect anomaly network traffics but also try to identify it with high accuracy.

NIDS Latency Analysis We perform the latency analysis based on M/M/1 model for software-NIDS and hardware-accelerated NIDS. [176]. We assume the packet arrivals following a Possion distribution with an average rate of α packets per second and the queued packed are processed with an exponential service rate of μ packet per second. The IDS module's utlization factor is defined as $\rho = \alpha/\mu$. In the steady state, the

Table 5.12: Intrusion Detection Accuracy comparison with other algorithms on NSL-KDD dataset

	Class				
Model	**Normal**	**DOS**	**Probe**	**R2L**	**Overall**
Proposed	95.82 %	76.21%	72.61%	0.25%	75.15%
SVM	92.8 %	74.7%	71.6%	12.3%	74.6%
MLP	93 %	71.2 %	60.1 %	0.001%	70.6%
Naive Bayes	85.8%	69.4%	32.8%	0.095%	70.5%

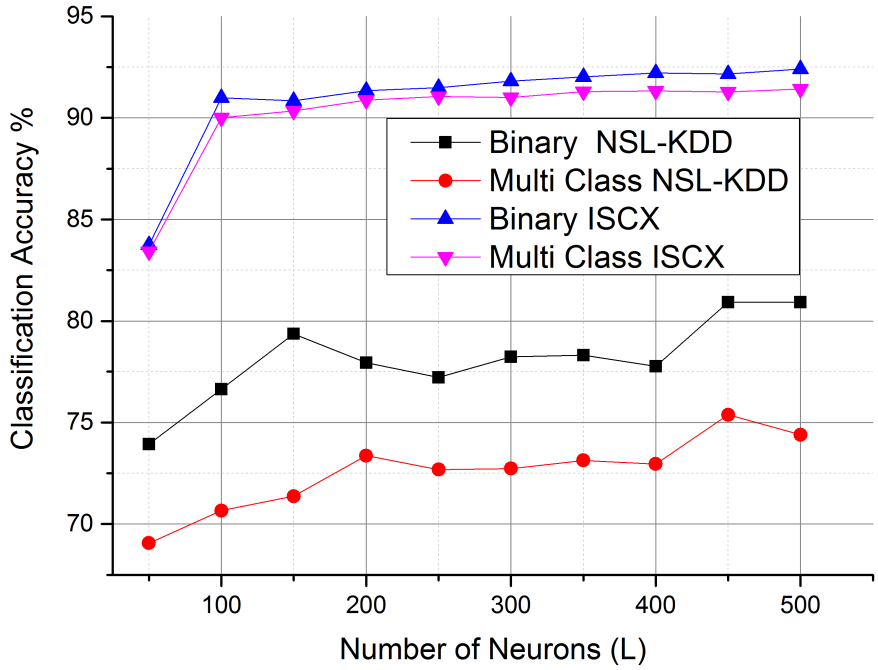


Figure 5.18: NIDS Classification Accuracy on NSL-KDD and ISCX datasets

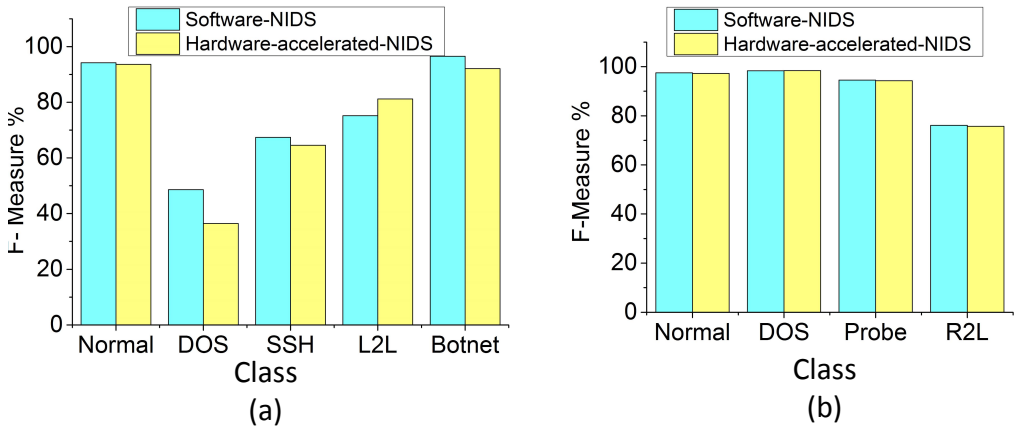


Figure 5.19: (a) ISCX 2012 MultiClass Classification (b) NSL-KDD MultiClass Classification

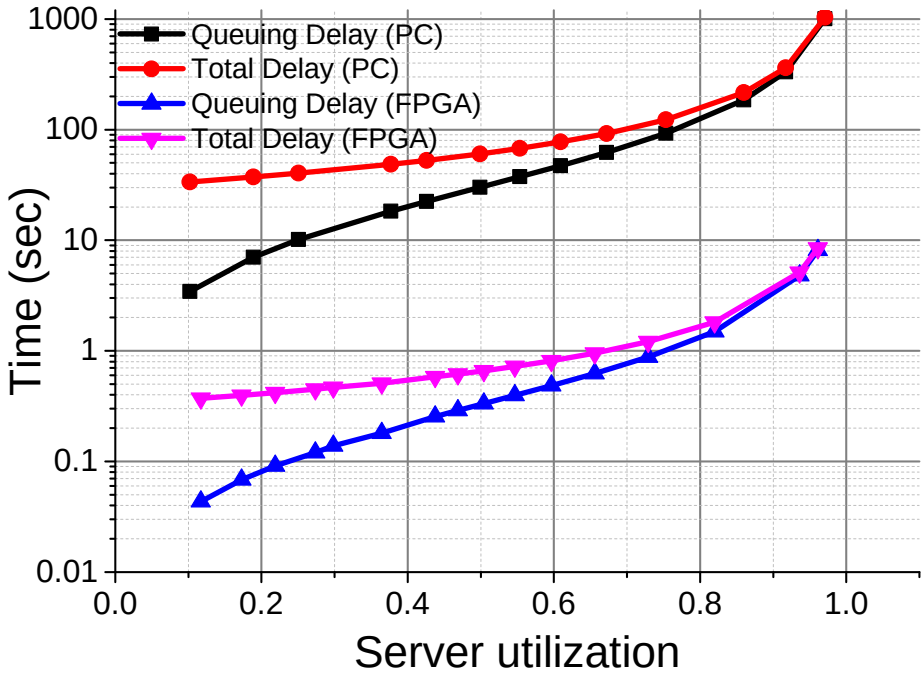


Figure 5.20: Total delay comparison between software-NIDS and hardware-accelerated NIDS

average queuing time W_{queue} can be calculated as $W_{queue} = \frac{\rho}{\mu(1-\rho)}$, whereas the total delay W_{Total} for each packet is given by (5.33). Fig. 5.20 shows the simulated result with an increasing arrival rate (packets/sec) given service time of software-NIDS and hardware-accelerated NIDS. The higher arrival rate is, the more server utilization will be. It clearly indicates that the total delay increases significantly for software-NIDS with various server utilization.

NIDS Platform Analysis In the experiment, the maximum throughput of proposed architecture is 12.68 GFLOPS with 128 parallelism for matrix multiplication under $50Mhz$ operations. This is slightly lower than theoretical maximum output of 12.8 GFLOPS, which can be calculated as $128 \times 50 \times 2 = 12.8$ GFLOPS. This is based on the multiplication of parallelism level, operating frequency and operands of each PE. The maximum input bandwidth is 409.6 Gbps and can be easily extended by higher parallelism and faster operating frequency.

To evaluate the energy consumption, we calculate the energy for a given implementation by multiplying the peak power consumption of the corresponding device with running time. Table 5.13 provides detailed comparisons between different platforms. Our proposed hardware-accelerated NIDS has the lowest power consumption ($0.85W$) compared to Embedded-NIDS ($2.5W$) and Software-NIDS implementation ($84W$). For

Table 5.13: Performance Comparison on ISCX 2012 Benchmark

Platform	Type	Format	Time	Power	Energy	Speed.	E. Imp.
Software	Train	Single	656s	84W	55104J	4.5×	449×
	Inference		30.24ms	84W	2540.2J	92.2×	9111×
Embedded	Train	Single	11183s	2.5W	27957.5J	77.4×	227.6×
	Inference		383ms	2.5W	957.6J	1168×	3434.7×
FPGA	Train	Single+ Fixed	144.5s	0.85W	122.8J	–	–
	Inference		0.328ms	0.85W	0.30J	–	–

training process, our accelerator has 4.5× and 77.4× speed-up for training compared to Software-NIDS and Embedded-NIDS. For inference process, it is mainly on matrix-vector multiplications. Our proposed method still has 92.2× and 1168× speed-up for inference compared to Software-NIDS and Embedded-NIDS implementations respectively. Furthermore, our proposed hardware-accelerated NIDS achieved around two orders of magnitude energy saving compared to other platforms in both inference and training process on benchmark ISCX-2012. In summary, our accelerator provides a low-power and low-latency performance of NIDS for the IoT network security.

5.6 Conclusion

To address dynamic ambient change in a large-scaled space, real-time and distributed data analytics is required on gateway network, which however has limited computing resources. In this chapter, we have discussed the application of distributed machine learning techniques for indoor data analytics on smart-gateway network. More specifically, this chapter investigates three applications, which are summarized as follows.

- An indoor positioning system is introduced based on the distributed neural network. One incremental ℓ_2-norm based solver is developed for learning collected WiFi-data at each gateway and is further fused for all gateways in the network to determine the location. Experimental results show that with 5 distributed gateways running in parallel for a $80m^2$ space, the proposed algorithm can achieve 50x and 38x improvement on inference and training time respectively when compared to support vector machine based data analytics with comparable positioning precision.

- An energy management system is presented based on distributed real-time data analytics. A solar energy allocation to reduce peak load of electricity power-grid is developed for smart buildings. The distributed real-time data analytics considers both occupant profile and energy profile using a fast machine learning

engine running on smart-gateway network without linkage to cloud. Experiment results show that the developed distributed real-time data analytics is 14.83% more accurate than the traditional support vector machine method. Compared to the static prediction, the short-term load prediction can achieve 51.94% more energy saving with 15.20% more peak load reduction.

- A real-time network intrusion detection is proposed based an online sequential machine learning hardware accelerator. A fast and low-power IoT NIDS can be achieved on FPGA based on optimized sequential learning algorithm to adapt to new threats. Furthermore, a single hidden layer feedforward neural network based learning algorithm is developed with an incremental least-squares solver realized on hardware. Experimental results on a single FPGA achieve a bandwidth of 409.6 Gbps with 4.5$\times$ and 77.4$\times$ speed-up compared to general CPU and embedded CPU. Our FPGA accelerator provides a low-power and low-latency intrusion detection performance for the IoT network security.

Experimental results show that such a computational intelligence technique can be compactly realized on the computational-resource limited smart-gateway networks, which is desirable to build a real cyber-physical system towards future smart homes, smart buildings, smart communities and further a smart city.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

The Internet of things (IoT) is to use networked objects with intelligence to improve resource efficiency. A typical IoT system can sense data from real-world, use sensed information to reason the environment and then perform the desired action. The intelligence of IoT systems comes from appropriate actions by reasoning the environmental data, which is mainly based on machine learning techniques. To have a real-time response to the dynamic ambient change, a machine learning accelerator on IoT edge devices is preferred since a centralized system suffers long latency of processing in the back end. However, IoT edge devices are resource-constrained and machine learning algorithms are computational intensive. Therefore, optimized machine learning algorithms, such as compact machine learning for less memory usage on IoT devices, is greatly needed. In this thesis, we explore the development of fast and compact machine learning accelerators by developing least-squares solver, tensor-solver and distributed-solver. Moreover, applications of such machine learning solver on IoT devices are also investigated. The main contribution of this thesis can be summarized as below.

From the fast machine learning perspective, the target is to perform fast learning on the neural network. This thesis proposes a least-squares-solver for a single hidden layer neural network. Furthermore, this thesis explores the CMOS FPGA based hardware accelerator and RRAM based hardware accelerator. To be more specific, firstly, an incremental and square-root-free Cholesky factorization algorithm is introduced with FPGA realization for training acceleration when analyzing the real-time sensed data. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable forecasting accuracy with an average speed-up of 4.56$\times$ and 89.05$\times$, when compared to x86 CPU and ARM CPU for inference respectively. Moreover , 450.2$\times$,

261.9× and 98.92× energy saving can be achieved comparing to x86 CPU, ARM CPU and GPU respectively. Secondly, a 3D multi-layer CMOS-RRAM accelerator architecture for incremental machine learning is proposed. By utilizing an incremental least-squares solver, the whole training process can be mapped to the 3D multi-layer CMOS-RRAM accelerator with significant speed-up and energy-efficiency improvement. Experiment results using the benchmark CIFAR-10 show that the proposed accelerator has 2.05× speed-up, 12.38× energy-saving and 1.28× area-saving compared to 3D-CMOS-ASIC hardware implementation; and 14.94× speed-up, 447.17× energy-saving and around 164.38× area-saving compared to CPU software implementation. Compared to GPU implementation, our work shows 3.07× speed-up and 162.86× energy-saving.

From the compact machine learning perspective, this thesis investigates a tensor-solver for deep neural networks with neural network compression. A layer-wise training of tensorized neural network (TNN) has been proposed to formulate multilayer neural network such that the weight matrix can be significantly compressed during training. By reshaping the multilayer neural network weight matrix into a high dimensional tensor with a low-rank approximation, significant network compression can be achieved with maintained accuracy. A corresponding layer-wise training is developed by a modified alternating least-squares (MALS) method without backward propagation (BP). TNN can provide state-of-the-art results on various benchmarks with significant compression. For MNIST benchmark, TNN shows 64× compression rate without accuracy drop. For CIFAR-10 benchmark, TNN shows that compression of 21.57× compression rate for fully-connected layers with 2.2% accuracy drop. In addition, a highly-parallel yet energy-efficient machine learning accelerator has been proposed for tensorized neural network. Simulation results using the benchmark MNIST show that the proposed accelerator has 1.283× speed-up, 4.276× energy-saving and 9.339× area-saving compared to 3D CMOS-ASIC implementation; and 6.37× speed-up and 2612× energy-saving compared to 2D CPU implementation. In addition, 14.85× model compression can be achieved by tensorization with acceptable accuracy loss.

From the large scaled IoT network perspective, this thesis proposes a distributed-solver on IoT devices. Furthermore, this thesis proposes a distributed neural network and sequential learning on the smart gateways for indoor positioning, energy management and IoT network security. For indoor positioning system, experimental results show that with multiple distributed gateways running in parallel, the proposed algorithm can achieve 50× and 38× speed-up during inference and training respectively with comparable positioning accuracy, when compared to traditional support vector machine (SVM) method. For energy management system, experiment results on real-life datasets have shown that the accuracy of the proposed energy prediction can be 14.83%

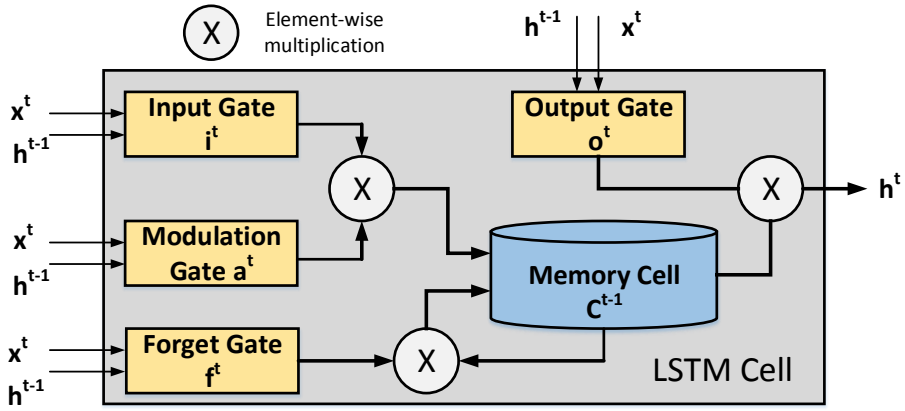


Figure 6.1: Long short-term memory (LSTM) cell

improvement comparing to SVM method. Moreover, the peak load from main electricity power-grid is reduced by 15.20% with 51.94% energy cost saving. For network intrusion detection of IoT systems, experimental results on a single FPGA achieve a bandwidth of 409.6 Gbps with 4.5× and 77.4× speed-up compared to general CPU and embedded CPU. Our FPGA accelerator provides a low-power and low-latency intrusion detection performance for the IoT network security.

6.2 Recommendations for Future Works

Based on above works, there are several recommended future works for this thesis.

The first recommended work is to explore the neural network compression of recurrent neural network (RNN). The RNN model has more redundancy and could be potentially greatly compressed. The Long Short-Term memory (LSTM) cell [177] in the RNN is shown as Fig. 6.1. This cell has 4 gates to actively select the information for the next layer. It will also memorize information in the memory cell C^{t-1}, where t is the time step. We denote x^t and h^{t-1} as the new input and last time step $t-1$ output. The four gates and the output from the cells are shown as

$$\begin{aligned} a^t &= tanh(W_c X^t + U_c H^{t-1}) \\ i^t &= \sigma(W_i X^t + U_i h^{t-1}) \\ f^t &= \sigma(W_f x^t + U_i h^{t-1}) \\ o^t &= \sigma(W_o x^t + U_o h^{t-1}) \end{aligned} \tag{6.1}$$

where W_c, W_f, W_i and W_0 are weight parameters for the modulation gate, forget gate, input gate and output gate respectively. U_c, U_f, U_i and U_0 are the weights for the last time step $t-1$ output respectively. In the LSTM network, instead of the weights from

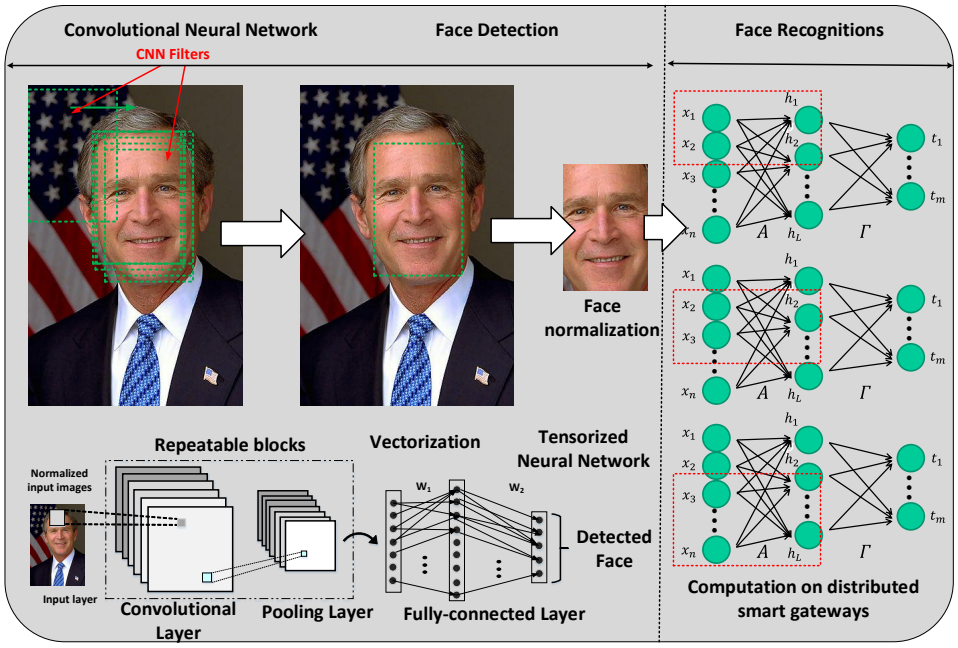


Figure 6.2: Proposed commutating algorithm on ASIC for face detection and recognition

one layer to the next layer, there are also weights for each gate. Therefore, we can perform the low-rank tensor-train decomposition to reduce the weight size to save the model size.

The second recommendation is to implement a tensorized neural network on chip for face detection. With the combination of the distributed neural network for face recognition, we can perform the whole face recognition process for the surveillance system. Although many algorithms have been developed for fast and robust face detection and recognition, it is still challenging to perform it in real-time applications such as video surveillance system. Therefore, it is interesting to develop a low power hardware accelerator for face recognitions at reasonable hardware cost with comparable recognition accuracy. As shown in Fig. 6.2, a tensorized neural network can effectively reduce the model size without hurting the performance and a distributed computation can utilize the computational resource on smart gateways. Kernels of conventional neural network can effectively extract features of one image to determine the face position. Then the normalized face will be input to the neural network for face recognitions. In the face recognition phase, we can perform the distributed computation on smart gateways. The common learning engine mentioned in Chapter 5 can be re-used for this purpose. We leave such application on IoT devices for future works.

For the overall picture of IoT systems, the bottleneck of massively deploying IoT devices in smart buildings comes from three limitations.

- First, the intelligence of IoT devices is not satisfying and still under development. To massively deploy IoT devices into smart homes requires low-power yet intelligent design. As such, it is important to optimize the machine learning algorithm to reduce the computation complexity as well as the memory required. Moreover, the design of IoT devices should also consider a customized architecture to accelerate the machine learning process with a low-power constraint. The hardware architecture optimized for machine learning algorithms remains an open problem to investigate.

- Second, the IoT networks still require more research to design the industrial communication standard. The technology adopted for communication between IoT devices should have a standard protocol with consideration of privacy and security. The integration between each IoT device has not been fully explored and utilized.

- Finally, the cost of IoT devices is still too high to be widely accepted by customers. For example, a smart power plug with Bluetooth communication costs 60 USD, which is too expensive comparing to the normal power plug [178]. The demand is created when a product provides better service than the existing product with a similar price.

The research community needs to further improve the IoT technology and collaborate with industries to develop low-cost intelligent IoT devices to improve human wellbeing.

Appendix A

Publication List

A.1 Book Chapter

1. **Hantao Huang**, Rai Suleman Khalid and Hao Yu, Distributed Machine Learning on Smart-Gateway Network Towards Real-time Indoor Data Analytics, Chapter in Data Science and Big Data: An Environment of Computational Intelligence, Witold Pedrycz and Shyi-Ming Chen (Editors), Springer 2017.

2. **Hantao Huang**, and Hao Yu, Least-squares-solver based machine learning accelerator for real-time data analytics in smart buildings, Chapter in Emerging Technology and Architecture for Big-data Analytics, Hao Yu, Chip Hong Chang and Anupam Chattopadhyay (Editors), Springer 2017

3. Leibin Ni, **Hantao Huang**,and Hao Yu, Distributed In-Memory Computing on Binary Memristor-Crossbar for Machine Learning, Chapter in Memristors, Memristive Devices and Systems, Sundarapandian Vaidyanathan and Christos Volos (Editors), Springer 2017.

A.2 Journal

1. **Hantao Huang**, Leibin Ni and Hao Yu, A Highly-parallel and Energy-efficient 3D Multi-layer CMOS-RRAM Accelerator for Tensorized Neural Network, *IEEE Transactions on Nanotechnology* (TNANO) 2017. (doi:10.1109/TNANO.2017.2732698)

2. **Hantao Huang**, and Hao Yu, Layer-Wise Training of Tensorized Neural Network, *IEEE Transactions on Neural Networks and Learning Systems* (TNNLS), 2017. (In Review)

3. **Hantao Huang**, Hang Xu, Yuehua Cai, Rai Suleman Khalid and Hao Yu, Distributed Machine Learning on Smart-gateway Network towards Real-time Smart-grid Energy Management with Behavior Cognition, *ACM TransaioDesign Automation of Electronic Systems* (TODAES) 2017. (In Review)

4. Dongjun Xu, Ningmei Yu, **Hantao Huang**, Sai Manoj P. D., and Hao Yu , Q-Learning based Voltage-swing Tuning and Compensation for 2.5D Memory-Logic Integration, *IEEE Design & Test*, 2017. (10.1109/MDAT.2017.2764075)

5. **Hantao Huang**, Yuehua Cai, Hang Xu and Hao Yu, A Multi-agent Minority-game based Demand-response Management of Smart Buildings towards Peak Load Reduction, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), 2016. (doi: 10.1109/TCAD.2016.2571847)

6. Leibin Ni, **Hantao Huang**, Zichuan Liu, Rajiv V. Joshi and Hao Yu, Distributed In-Memory Computing on Binary RRAM Crossbar, *ACM Journal on Emerging Technologies in Computing System* (JETC), 2017. (doi: 10.1145/2996192)

7. Sai Manoj P.D., Hao Yu, **Hantao Huang** and Dongjun Xu, A Q-Learning based Self-adaptive I/O Communication for 2.5D Integrated Many-core Microprocessor and Memory, *IEEE Transactions on Computers* (TC), vol.65, no.4, pp1185-1196, April 2016. (doi:10.1109/TC.2015.2439255)

A.3 Conference

1. **Huang, Hantao**, Rai Suleman Khalid, Wenye Liu, Hao Yu "A fast online sequential learning accelerator for IoT network intrusion detection: work-in-progress." IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion. ACM, 2017.

2. **Hantao Huang**, Leibin Ni and Hao Yu, TNN: An Energy-efficient Machine Learning Accelerator on 3D CMOS-RRAM for Tensorized Neural Network, IEEE International System-on-chip conference (ISOCC), 2017 .

3. **Hantao Huang**, Rai Sulman Khalid and Hao Yu, IoT Network Intrusion Detection by Online Sequential Machine Learning Accelerator, ACM.IEEE Design Automation Conference (DAC), 2017 (WIP).

4. **Hantao Huang**,, Leibin Ni, Yuhao Wang, Hao Yu, Zongwei Wang, Yimao Cai and Ru Huang, A 3D Multi-layer CMOS-RRAM Accelerator for Neural Network, IEEE International 3D System Integration Conference (3DIC), November 2016.

5. Hang Xu, **Hantao Huang**, Rai Suleman Khalid and Hao Yu, Distributed Machine Learning based Smart-grid Energy Management with Behaviour Cognition, IEEE International Conference on Smart Grid Communications (SmartGrid Comm), November 2016.

6. **Hantao Huang**, Leibin Ni and Hao Yu, A 3D Multi-layer CMOS-RRAM Accelerator for Multi-layer Machine Learning, IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT) (Invited), October 2016.

7. Wei Pang, **Hantao Huang**, Fengwei An, and Hao Yu, "Low-power and Real-time Computer Vision On-chip", IEEE System-on-Chip Conference (ISOCC) (Invited), Korea, October 2016

8. Leibin Ni, **Hantao Huang** and Hao Yu, A Memristor Network with Coupled Oscillator and Crossbar towards L2-norm based Machine Learning, ACM/IEEE International Symposium on Nanoscale Architectures (NanoArch), July 2016

9. Leibin Ni, **Hantao Huang** and Hao Yu, "On-line Machine Learning Accelerator on Digital RRAM-Crossbar", IEEE International Symposium on Circuits and Systems (ISCAS) (Invited Special Session), Montreal, May 2016.

10. **Hantao Huang**, Hao Yu, Chen Zhuo and Fengbo Ren, "A Compressive-sensing based Inference Vehicle for 3D TSV Pre-bond and Post-bond Inference Data", *ACM International Symposium on Physical Design* (ISPD) 2016.

11. **Hantao Huang**, Yuehua Cai and Hao Yu, "Distributed-neuron-network based Machine Learning on Smart-gateway Network towards Real-time Indoor Data Analytics", *ACM/IEEE Design Automation and Test Conference in Europe* (DATE) 2016.

12. Sai Manoj P. D, Kanwen Wang, **Hantao Huang** and Hao Yu, "Smart I/Os: A Data-pattern Aware 2.5D Interconnect with Space-Time Multiplexing", *ACM/IEEE System Level Interconnect Prediction* (SLIP), June 2015.

13. Yuhao Wang, **Hantao Huang**, Leibin Ni, Hao Yu, Mei Yan, Chuliang Wen, Wei Yang and Junfeng Zhao "An Energy-efficient Non-volatile In-Memory DataAnalytic for Sparse-represented Face Recognition", *ACM/IEEE Design Automation and Test Conference in Europe* (DATE), March 2015.

14. **Hantao Huang**, Sai Manoj P.D., Dongjun Xu, Hao Yu, and Zhigang Hao, "Reinforcement Learning based Self-adaptive Voltage-swing Adjustment of Through-silicon Interposer I/Os for Many-core Microprocessor and Memory Communication", *ACM/IEEE International Conference on Computer-Aided-Design* (ICCAD), November 2014

15. Dongjun Xu, Sai Manoj P. D., **Hantao Huang**, Ningmei Yu, and Hao Yu, "An Energy-efficient 2.5D Through-silicon Interposer I/O with Self-adaptive Adjustment of Output-voltage Swing", *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), August 2014.

Bibliography

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, Lake Tahoe, Nevada, 2012, pp. 1223–1231.

[2] M. E. Hussein, M. Torki, M. A. Gowayyed, and M. El-Saban, "Human action recognition using a temporal hierarchy of covariance descriptors on 3D joint locations," in *International Joint Conference on Artificial Intelligence*, Menlo Park, California, 2013, pp. 2466–2472.

[3] H. Yahmadi, "Internet of things," Online: https://community.icann.org/download/attachments/52890780/Internet%20of%20Things%20%28IoT%29%20-%20Hafedh%20Alyahmadi%20-%20May%2029%2C%202015.pdf?version=1&modificationDate=1432982332000&api=v2, accessed: 2017-05-30.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[5] H. Huang, Y. Cai, and H. Yu, "Distributed-neuron-network based machine learning on smart-gateway network towards real-time indoor data analytics," in *Conference on Design, Automation & Test in Europe*, Dresden, Germany, 2016, pp. 720–725.

[6] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, Sardinia, Italy, 2010, pp. 249–256.

[8] S. Han, 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.

[9] S. Han and et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *International Symposium on Field-Programmable Gate Arrays*, Monterey, California, 2017, pp. 75–84.

[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[11] A. Davis and I. Arel, "Low-rank approximations for conditional feedforward computation in deep neural networks," *arXiv preprint arXiv:1312.4461*, 2013.

[12] I. Hubara, D. Soudry, and R. E. Yaniv, "Binarized neural networks," *arXiv preprint arXiv:1602.02505*, 2016.

[13] I. Micron Technology, "Breakthrough nonvolatile memory technology," Online: http://www.micron.com/about/emerging-technologies/3d-xpoint-technology/, accessed: 2018-01-04.

[14] S. Yu and et al., "3D vertical RRAM-scaling limit analysis and demonstration of 3D array operation," in *Symposium on VLSI Technology and Circuits*, Kyoto, Japan, 2013, pp. 158–159.

[15] M. Hung, "Leading the IoT," Online: https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf, accessed: 2018-02-09.

[16] H. Liu, H. Darabi, P. Banerjee, and J. Liu, "Survey of wireless indoor positioning techniques and systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 37, no. 6, pp. 1067–1080, 2007.

[17] C. Yang and H.-R. Shao, "Wi-Fi-based indoor positioning," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 150–157, 2015.

[18] A. M. Hossain and W.-S. Soh, "A survey of calibration-free indoor positioning systems," *Computer Communications*, vol. 66, pp. 1–13, 2015.

[19] S. He and S.-H. G. Chan, "Wi-Fi fingerprint-based indoor positioning: Recent advances and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 466–490, 2016.

[20] J. Xu, W. Liu, F. Lang, Y. Zhang, and C. Wang, "Distance measurement model based on rssi in WSN," *Wireless Sensor Network*, vol. 2, no. 08, p. 606, 2010.

[21] J. Torres-Solis, T. H. Falk, and T. Chau, *A review of indoor localization technologies: towards navigational assistance for topographical disorientation*. INTECH Publisher, 2010.

[22] J. Janicka and J. Rapinski, "Application of RSSI based navigation in indoor positioning," in *Geodetic Congress (Geomatics), Baltic*, Gdansk, Poland, 2016, pp. 45–50.

[23] Y. Cai, S. K. Rai, and H. Yu, "Indoor positioning by distributed machine-learning based data analytics on smart gateway network," in *International Conference on Indoor Positioning and Indoor Navigation*, Alberta, Canada, 2015, pp. 1–8.

[24] D. Li and D. L. Lee, "A topology-based semantic location model for indoor applications," in *International Workshop on Security and Privacy in GIS and LBS*, Irvine, California, 2008, pp. 6:1–6:10.

[25] M. Altini, D. Brunelli, E. Farella, and L. Benini, "Bluetooth indoor localization with multiple neural networks," in *International Symposium on Wireless Pervasive Computing*, Modena, Italy, 2010, pp. 295–300.

[26] L. D. Harvey, *Energy and the New Reality 1: Energy Efficiency and the Demand for Energy Services*. Routledge, 2010.

[27] N. Lu and et.al., "The temperature sensitivity of the residential load and commercial building load," in *Power Energy Society General Meeting*, Alberta, Canada, 2009, pp. 1–7.

[28] J. Kleissl and Y. Agarwal, "Cyber-physical energy systems: focus on smart buildings," in *Proceeding of Design Automation Conference*, San Francisco, California, 2010, pp. 749–754.

[29] F. Zhang, H. Deng, R. Margolis, and J. Su, "Analysis of distributed-generation photo voltaic deployment, installation time and cost, market barriers, and policies in china," *Energy Policy*, vol. 81, pp. 43–55, 2015.

[30] D. B. Richardson and L. Harvey, "Strategies for correlating solar PV array production with electricity demand," *Renewable Energy*, vol. 76, pp. 432–440, 2015.

[31] B. A. Hoverstad, A. Tidemann, H. Langseth, and P. Ozturk, "Short-term load forecasting with seasonal decomposition using evolution for parameter tuning," *IEEE Transactions on Smart Grid*, vol. 6, no. 4, pp. 1904–1913, 2015.

[32] H. S. Hippert, C. E. Pedreira, and R. C. Souza, "Neural networks for short-term load forecasting: A review and evaluation," *IEEE Transactions on power systems*, vol. 16, no. 1, pp. 44–55, 2001.

[33] Q. Cheng, J. Yao, H. Wu, S. Chen, C. Liu, and P. Yao, "Short-term load forecasting with weather component based on improved extreme learning machine," in *Chinese Automation Congress*, Hunan, China, 2013, pp. 316–321.

[34] B. J. Johnson, M. R. Starke, O. A. Abdelaziz, R. K. Jackson, and L. M. Tolbert, "A method for modeling household occupant behavior to simulate residential energy consumption," in *Innovative Smart Grid Technologies Conference*, Washington, DC, USA, 2014, pp. 1–5.

[35] C. Sandels, J. Widén, and L. Nordström, "Forecasting household consumer electricity load profiles with a combined physical and behavioral approach," *Applied Energy*, vol. 131, pp. 267–278, 2014.

[36] S. Chen, T. Liu, Y. Zhou, C. Shen, F. Gao, Y. Che, and Z. Xu, "SHE: Smart home energy management system based on social and motion behavior cognition," in *International Conference on Smart Grid Communications*, Miamimiami, Floridia, 2015, pp. 859–864.

[37] H. Polinder and et.al, *International Energy Agency program on Buildings & Community. Annex 43-Occupants behavior and modeling*. International Energy Agency, 2013.

[38] E. Viegas, A. O. Santin, A. França, R. Jasinski, V. A. Pedroni, and L. S. Oliveira, "Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 163–177, 2017.

[39] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems(idps)," *NIST special publication*, vol. 800, pp. 94–96, 2007.

[40] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *USENIX Systems Administration Conference*, Seattle, Washington, 1999, pp. 229–238.

[41] N. B. Guinde and S. G. Ziavras, "Efficient hardware support for pattern matching in network intrusion detection," *computers & security*, vol. 29, no. 7, pp. 756–769, 2010.

[42] K. Hwang, M. Cai, Y. Chen, and M. Qin, "Hybrid intrusion detection with weighted signature generation over anomalous Internet episodes," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 1, 2007.

[43] S. Theodoridis, A. Pikrakis, K. Koutroumbas, and D. Cavouras, *Introduction to Pattern Recognition: A Matlab Approach*. Academic Press, 2010.

[44] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[45] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[46] D. Amodei and et.al., "Deep speech 2: End-to-end speech recognition in English and mandarin," in *International Conference on Machine Learning*, New York City, 2016, pp. 173–182.

[47] Y. Bengio *et al.*, "Learning deep architectures for AI," *Foundations and trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[48] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, Caesars Palace, Nevada, 2016, pp. 770–778.

[49] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[50] A. Hadoop, Online: http://hadoop.apache.org, access: 2016-11-02.

[51] M. Zaharia and et.al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.

[52] "Apache spark hardware provisioning," Online: https://spark.apache.org/docs/0.9.0/hardware-provisioning.html, accessed: 2017-05-03.

[53] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[55] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42 tops/w deep convolutional neural network recognition processor for intelligent IoT systems," in *International Solid-State Circuits Conference*, San Francisco, California, 2016, pp. 264–265.

[56] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[57] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International Conference on Artificial Neural Networks*, Hamburg, Germany, 2014, pp. 281–290.

[58] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays*, Monterey, California, 2015, pp. 161–170.

[59] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.

[60] L. Xia and et.al., "Technological exploration of RRAM crossbar array for matrix-vector multiplication," *Journal of Computer Science and Technology*, vol. 31, no. 1, pp. 3–19, 2016.

[61] L. Ni and et.al., "An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary RRAM crossbar," in *Asia and South Pacific Design Automation Conference*, Macao, China, 2016, pp. 280–285.

[62] Y. Wang and et.al., "Energy efficient RRAM spiking neural network for real time classification," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, Pittsburgh, Pennsylvania, 2015, pp. 189–194.

[63] L. Ni, H. Huang, and H. Yu, "On-line machine learning accelerator on digital rram-crossbar," in *International Symposium on Circuits and Systems*, Montreal, Canada, 2016, pp. 113–116.

[64] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick." in *International Conference on Machine Learning*, Lille, France, 2015, pp. 2285–2294.

[65] W. Chen, J. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing convolutional neural networks in the frequency domain," in *International Conference on Knowledge Discovery and Data Mining*, San Francisco, California, 2016, pp. 1475–1484.

[66] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *arXiv preprint arXiv:1609.07061*, 2016.

[67] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, New York City, 2016, pp. 2849–2858.

[68] S. Abid, F. Fnaiech, and M. Najim, "A new neural network pruning method based on the singular value decomposition and the weight initialization," in *European Signal Processing Conference*, Toulouse, France, 2002, pp. 1–4.

[69] C. Bucilu, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *International conference on Knowledge discovery and data mining*, Philadelphia,Pennsylvania, 2006, pp. 535–541.

[70] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[71] P. Nakkiran, R. Alvarez, R. Prabhavalkar, and C. Parada, "Compressing deep neural networks using a rank-constrained topology," in *Sixteenth Annual Conference of the International Speech Communication Association*, Dresden, Germany, 2015.

[72] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, Montreal Canada, 2015, pp. 442–450.

[73] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, Lake Tahoe, Nevada, 2013, pp. 2148–2156.

[74] S. Holtz, T. Rohwedder, and R. Schneider, "The alternating linear scheme for tensor optimization in the tensor train format," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. A683–A713, 2012.

[75] D. Silver and et.al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[76] M. Li and et.al., "Scaling distributed machine learning with the parameter server," in *USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, Colorado, 2014, pp. 583–598.

[77] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *International Joint Conference on Neural Networks*, Washington, DC, 1989, pp. 593–605.

[78] B. Igelnik, B. Igelnik, and J. M. Zurada, *Efficiency and Scalability Methods for Computational Intellect*, 1st ed. IGI Global, 2013.

[79] J. Qiu and et.al., "Going deeper with embedded fpga platform for convolutional neural network," in *International Symposium on Field-Programmable Gate Arrays*, Monterey, California, 2016, pp. 26–35.

[80] F. Ren and D. Markovi, "A configurable 12237 kS/s 12.8 mw sparse-approximation engine for mobile data aggregation of compressively sampled physiological signals," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 68–78, Jan 2016.

[81] Q. Wang, P. Li, and Y. Kim, "A parallel digital VLSI architecture for integrated support vector machine training and classification," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 23, no. 8, pp. 1471–1484, 2015.

[82] G. B. Huang, Q. Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[83] Y. Wang and et.al., "An energy-efficient nonvolatile in-memory computing architecture for extreme learning machine by domain-wall nanowire devices," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 998–1012, 2015.

[84] S. Decherchi, P. Gastaldo, A. Leoncini, and R. Zunino, "Efficient digital implementation of extreme learning machines for classification," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 8, pp. 496–500, 2012.

[85] Y.-H. Pao, G.-H. Park, and D. J. Sobajic, "Backpropagation, Part IV Learning and generalization characteristics of the random vector functional-link net," *Neurocomputing*, vol. 6, no. 2, pp. 163 – 180, 1994.

[86] M. D. Martino, S. Fanelli, and M. Protasi, "A new improved online algorithm for multi-decisional problems based on MLP-networks using a limited amount of information," in *International Joint Conference on Neural Networks*, Nagoya, Japan, 1993, pp. 617–620.

[87] L. N. Trefethen and D. Bau III, *Numerical linear algebra*. Siam, 1997, vol. 50.

[88] A. Krishnamoorthy and D. Menon, "Matrix inversion using Cholesky decomposition," *arXiv preprint arXiv:1111.4144*, 2011.

[89] "Adm-pcie-7v3," Online: http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3, accessed: 13-June-2016.

[90] L. Ni and et al., "An energy-efficient matrix multiplication accelerator by distributed in-memory computing on binary RRAM crossbar," in *Asia and South Pacific Design Automation Conference*, Macao, China, 2016, pp. 280–285.

[91] K. H. Kim and et.al., "A functional hybrid memristor crossbar-array/CMOS system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2011.

[92] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[93] L. Xia and et.al., "Technological exploration of RRAM crossbar array for matrix-vector multiplication," *Journal of Computer Science and Technology*, vol. 31, no. 1, pp. 3–19, 2016.

[94] P. Y. Chen and et al., "Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip," in *Design, Automation & Test in Europe Conference & Exhibition*, Grenoble, France, 2015.

[95] R. O. Topaloglu, *More than Moore technologies for next generation computer design*. Springer, 2015.

[96] Y. Y. Liauw, Z. Zhang, W. Kim, A. El Gamal, and S. S. Wong, "Nonvolatile 3D-FPGA with monolithically stacked rram-based configuration memory," in *IEEE International Solid-State Circuits Conference*, San Francisco, California, 2012.

[97] Y.-C. Chen, W. Wang, H. Li, and W. Zhang, "Non-volatile 3D stacking RRAM-based FPGA," in *IEEE International Conference on Field Programmable Logic and Applications*, Oslo, Norway, 2012.

[98] "Beagleboard-xm," Online: http://beagleboard.org/beagleboard-xm, 2016.

[99] M. Lichman, "UCI machine learning repository," Online: http://archive.ics.uci.edu/ml, 2013.

[100] J. A. Suykens and et.al., *Least squares support vector machines*. World Scientific, 2002, vol. 4.

[101] P. Franzon and et. al., "Computing in 3D," in *Custom Integrated Circuits Conference*, San Jose, California, Sept 2015, pp. 1–6.

[102] D. H. Kim, K. Athikulwongse, and S. K. Lim, "Study of through-silicon-via impact on the 3-D stacked IC layout," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 5, pp. 862–874, 2013.

[103] H. Lee and et al., "Low power and high speed bipolar switching with a thin reactive Ti buffer layer in robust HfO2 based RRAM," in *IEEE Electron Devices Meeting,*, 2008.

[104] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[105] S. Han and et. al., "DSD: Regularizing deep neural networks with dense-sparse-dense training flow," *arXiv preprint arXiv:1607.04381*, 2016.

[106] Z. Liu, Y. Li, F. Ren, and H. Yu, "A binary convolutional encoder-decoder network for real-time natural scene text processing," *arXiv preprint arXiv:1612.03630*, 2016.

[107] Y. Wang and et.al., "An energy-efficient nonvolatile in-memory computing architecture for extreme learning machine by domain-wall nanowire devices," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 998–1012, 2015.

[108] Y. Wang, C. Zhang, R. Nadipalli, H. Yu, and R. Weerasekera, "Design exploration of 3D stacked non-volatile memory by conductive bridge based crossbar," in *IEEE International Conference on 3D System Integration*, Osaka, Japan, 2012.

[109] Y. Wang, X. Li, K. Xu, F. Ren, and H. Yu, "Data-driven sampling matrix boolean optimization for energy-efficient biomedical signal acquisition by compressive sensing," *IEEE transactions on biomedical circuits and systems*, vol. 11, no. 2, pp. 255–266, 2017.

[110] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[111] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[112] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural network design*. PWS publishing company Boston, 1996, vol. 20.

[113] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle *et al.*, "Greedy layer-wise training of deep networks," *Advances in neural information processing systems*, vol. 19, p. 153, 2007.

[114] J. Tang, C. Deng, and G.-B. Huang, "Extreme learning machine for multilayer perceptron," *IEEE transactions on neural networks and learning systems*, vol. 27, no. 4, pp. 809–821, 2016.

[115] A. Rosenberg, ": Linear regression with regularization," Online: url-http://eniac.cs.qc.cuny.edu/andrew/gcml/lecture5.pdf, February 2009.

[116] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[117] L. L. C. Kasun, Y. Yang, G.-B. Huang, and Z. Zhang, "Dimension reduction with extreme learning machine," *IEEE Transactions on Image Processing*, vol. 25, no. 8, pp. 3906–3918, 2016.

[118] I. V. Oseledets and S. Dolgov, "Solution of linear systems and matrix inversion in the tt-format," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. A2718–A2739, 2012.

[119] Y. Wang, H. Yu, and W. Zhang, "Nonvolatile CBRAM-crossbar-based 3D-integrated hybrid memory for data retention," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 957–970, 2014.

[120] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "DESTINY: A Tool for Modeling Emerging 3D NVM and eDRAM caches," in *Design, Automation & Test in Europe Conference*, Grenoble, France, 2015, pp. 1543–1546.

[121] K. Chen and et.al., "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Dresden, Germany, 2012, pp. 33–38.

[122] L. Ni, H. Huang, Z. Liu, R. V. Joshi, and H. Yu, "Distributed in-memory computing on binary RRAM crossbar," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 36, 2017.

[123] A. Vedaldi and K. Lenc, "Matconvnet: Convolutional neural networks for MATLAB," in *International conference on Multimedia*, Brisbane, Australia, 2015, pp. 689–692.

[124] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[125] J. Xue, J. Li, and Y. Gong, "Restructuring of deep neural network acoustic models with singular value decomposition." in *Annual Conference of the International Speech Communication Association*, Lyon, France, 2013, pp. 2365–2369.

[126] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "High-performance neural networks for visual object classification," *arXiv preprint arXiv:1102.0183*, 2011.

[127] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," 2014.

[128] S. Fothergill, H. Mentis, P. Kohli, and S. Nowozin, "Instructing people for training gestural interactive systems," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Austin, Texas, 2012, pp. 1737–1746.

[129] W. Li, Z. Zhang, and Z. Liu, "Action recognition based on a bag of 3D points," in *Computer Vision and Pattern Recognition Workshops*, San Francisco, California, 2010, pp. 9–14.

[130] S. Yang, C. Yuan, W. Hu, and X. Ding, "A hierarchical model based on latent dirichlet allocation for action recognition," in *International Conference on Pattern Recognition*, Stockholm, Sweden, 2014, pp. 2613–2618.

[131] L. Zhou, W. Li, Y. Zhang, P. Ogunbona, D. T. Nguyen, and H. Zhang, "Discriminative key pose extraction using extended lc-ksvd for action recognition," in *International Conference on Digital lmage Computing: Techniques and Applications*, New South Wales, Australia, 2014, pp. 1–8.

[132] P. Wang, Z. Li, Y. Hou, and W. Li, "Action recognition based on joint trajectory maps using convolutional neural networks," in *ACM Multimedia Conference*, Amsterdam, The Netherlands, 2016, pp. 102–106.

[133] Y. Annadani, D. Rakshith, and S. Biswas, "Sliding dictionary based sparse representation for action recognition," *arXiv preprint arXiv:1611.00218*, 2016.

[134] J. Martens and I. Sutskever, "Learning recurrent neural networks with hessian-free optimization," in *International Conference on Machine Learning*, Bellevue, Washington, 2011, pp. 1033–1040.

[135] L. Xia, C.-C. Chen, and J. Aggarwal, "View invariant human action recognition using histograms of 3D joints," in *Computer Vision and Pattern Recognition Workshops*, Providence, Rhode Island, 2012, pp. 20–27.

[136] J. Wang, Z. Liu, J. Chorowski, Z. Chen, and Y. Wu, "Robust 3D action recognition with random occupancy patterns," in *Computer vision (ECCV)*. Springer, 2012, pp. 872–885.

[137] B. B. Amor, J. Su, and A. Srivastava, "Action recognition using rate-invariant analysis of skeletal shape trajectories," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 1, pp. 1–13, 2016.

[138] "GPU specs," Online: http://www.nvidia.com/object/workstation-solutions.html, accessed: 2017-03-30.

[139] B. Govoreanu, G. Kar, and et.al., "10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation," in *International Electron Devices Meeting*, Washington, DC, 2011, pp. 31–6.

[140] W. Fei, H. Yu, W. Zhang, and K. S. Yeo, "Design exploration of hybrid CMOS and memristor circuit by new modified nodal analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 6, pp. 1012–1025, 2012.

[141] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[142] "8-bit inference with tensorrt," Online: http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf, accessed: 2018-01-04.

[143] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary neural networks with fine-grained quantization," *arXiv preprint arXiv:1705.01462*, 2017.

[144] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277–298, 2009.

[145] M. O. Ergin, V. Handziski, and A. Wolisz, "Node sequence discovery in wireless sensor networks," in *International Conference on Distributed Computing in Sensor Systems*, Cambridge, Massachusetts, 2013, pp. 394–401.

[146] J. P. Anderson, "Computer security threat monitoring and surveillance," James P. Anderson Company, Fort Washington, Pennsylvania, Tech. Rep., 1980.

[147] E. Nyakundi, "Using support vector machines in anomaly intrusion detection," Ph.D. dissertation, The University of Guelph, 2015.

[148] D. J. Weller-Fahy, B. J. Borghetti, and A. A. Sodemann, "A survey of distance and similarity measures used within network intrusion anomaly detection," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 70–91, 2015.

[149] C.-C. Sun, C.-C. Liu, and J. Xie, "Cyber-physical system security of a power grid: State-of-the-art," *Electronics*, vol. 5, no. 3, p. 40, 2016.

[150] Y. Zhang, L. Wang, W. Sun, R. C. Green II, and M. Alam, "Distributed intrusion detection system in a multi-layer network architecture of smart grids," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 796–808, 2011.

[151] D. L. Donoho, "Compressed sensing," *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[152] N.-Y. Liang, G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A fast and accurate online sequential learning algorithm for feedforward networks," *IEEE Transactions on Neural Networks*, vol. 17, no. 6, pp. 1411–1423, 2006.

[153] L. L. Minku, A. P. White, and X. Yao, "The impact of diversity on online ensemble learning in the presence of concept drift," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 5, pp. 730–742, 2010.

[154] D. J. Miller and L. Yan, "Critic-driven ensemble classification," *IEEE Transactions on Signal Processing*, vol. 47, no. 10, pp. 2833–2844, 1999.

[155] C. Drane, M. Macnaughtan, and C. Scott, "Positioning GSM telephones," *IEEE Communications Magazine*, vol. 36, no. 4, pp. 46–54, 1998.

[156] P. Bahl and V. N. Padmanabhan, "RADAR: An in-building RF-based user location and tracking system," in *Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, Tel Aviv, Israel, 2000, pp. 775–784.

[157] M. Brunato and R. Battiti, "Statistical learning theory for location fingerprinting in wireless LANs," *Computer Networks*, vol. 47, no. 6, pp. 825–845, 2005.

[158] H. Mehmood, N. K. Tripathi, and T. Tipdecho, "Indoor positioning system using artificial neural network," *Journal of Computer science*, vol. 6, no. 10, p. 1219, 2010.

[159] M. Stella, M. Russo, and D. Begusic, "Location determination in indoor environment based on RSS fingerprinting and artificial neural network," in *International Conference on Telecommunications*, Zagreb, Croatia, 2007, pp. 301–306.

[160] H. Huang, Y. Cai, H. Xu, and H. Yu, "A multi-agent minority-game based demand-response management of smart buildings towards peak load reduction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 573–585, April 2017.

[161] N. Hampshire, "Pricing strategy for new hampshire electric co-op," Online: http://www.oca.nh.gov/Advisory%20Board/ArchivedMinutes/20150727Mtg/NHEC%20Presentation%20re%20AMI%20DP%20Prepaid%207-27-15.pdf, 2015, accessed at June 2015.

[162] J. Wang, K. Li, Q. Lv, H. Zhou, and L. Shang, "Hybrid energy storage system integration for vehicles," in *International symposium on low power electronics and design*, Richardson, Texas, 2010, pp. 369–374.

[163] C. Zhang, W. Wu, H. Huang, and H. Yu, "Fair energy resource allocation by minority game algorithm for smart buildings," in *Design Automation Conference in Europe*, Dresden, Germany, 2012.

[164] I. Richardson, M. Thomson, and D. Infield, "A high-resolution domestic building occupancy model for energy demand simulations," *Energy and buildings*, vol. 40, no. 8, pp. 1560–1566, 2008.

[165] N. Ye, Y. Liu, and Y. Wang, "Short-term power load forecasting based on SVM," in *World Automation Congress*, Puerto Vallarta, Mexico, 2012.

[166] F. Kuang, S. Zhang, Z. Jin, and W. Xu, "A novel SVM by combining kernel principal component analysis and improved chaotic particle swarm optimization for intrusion detection," *Soft Computing*, vol. 19, no. 5, pp. 1187–1199, 2015.

[167] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht, "Smart*: An open data set and tools for enabling research in sustainable homes," in *Proceedings of the 2012 Workshop on Data Mining Applications in Sustainability*, Beijing, China, 2012, pp. 112–118.

[168] S. E. Generation, "Solar energy generation profiles, elizabeth city state university," Online: http://rredc.nrel.gov/solar/new_data/confrrm/ec/, 2016, [Online; accessed Feb-2016].

[169] F. Maciá-Pérez and et.al., "Network intrusion detection system embedded on a smart sensor," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 722–732, 2011.

[170] J. Cannady, "Next generation intrusion detection: Autonomous reinforcement learning of network attacks," in *National information systems security conference*, Baltimore, Maryland, 2000, pp. 1–12.

[171] Y. Li, C. Zhang, and L. Yang, "The research of ami intrusion detection method using elm in smart grid," *International Journal of Security and Its Applications*, vol. 10, no. 5, pp. 283–296, 2016.

[172] P. Egli and A. Netmodule, "Susceptibility of wireless devices to denial of service attacks," *White paper, Netmodule AG, Niederwangen, Switzerland*, 2006.

[173] E. Hodo and et. al., "Threat analysis of IoT networks using artificial neural network intrusion detection system," in *International Symposium on Networks, Computers and Communications*, Yasmine Hammamet, Tunisia, 2016, pp. 1–6.

[174] "ISCX 2012 dataset," Online: http://www.unb.ca/research/iscx/dataset/iscx-IDS-dataset.html, accessed: 2017-06-04.

[175] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "NSL-kdd dataset," Online: http://www.iscx.ca/NSL-KDD, 2012, accessed: 2017-06-04.

[176] N. Tsikoudis, A. Papadogiannakis, and E. P. Markatos, "LEoNIDS: a Low-latency and Energy-efficient Network-level Intrusion Detection System," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 1, pp. 142–155, 2016.

[177] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent neural network regularization,” *arXiv preprint arXiv:1409.2329*, 2014.

[178] “Bluetooth power plug,” Online: https://www.cnet.com/news/smart-switch-buying-guide/, accessed: 2018-01-04.